



BEST AVAILABLE COPY

Attorney's Docket No.: 16441-012002

THE UNITED STATES PATENT AND TRADEMARK OFFICE

Applicant : John Christian Hermansen et al. Art Unit : 2641
 Serial No. : 10/055,178 Examiner : Joon H. Hwang
 Filed : January 25, 2002
 Title : SYSTEM AND METHOD FOR ADAPTIVE MULTI-CULTURAL SEARCHING
 AND MATCHING OF PERSONAL NAMES

MAIL STOP AMENDMENT

Commissioner for Patents
 P.O. Box 1450
 Alexandria, VA 22313-1450

DECLARATION IN RESPONSE TO THE REQUIREMENT OF 37 CFR § 1.105

This declaration is prepared in response to the 37 C.F.R. § 1.105 requirement in the Office Action mailed on October 21, 2004.

I, the Declarant, am one of the named inventors on U.S. Patent Application No. 10/055,178, in which an office action was mailed on October 21, 2004. The Office Action includes a 105 requirement relating to (i) PC-NAS and (ii) products and services of Language Analysis Systems, Inc. ("LAS").

The present application, as-filed, included a paragraph asserting to describe PC-NAS, stating in part that "The assignee has developed a software program known as PC-NAS. An early version of this program was incorporated into a government computer system more than one year before the priority date of this application." Specification at page 5, lines 11-13. This statement is inaccurate for at least the reason that PC-NAS was not incorporated into a government computer system more than one year before the priority date of this application. Because the statement was inaccurate, I, through my patent attorneys, removed the PC-NAS paragraph from the specification in an amendment filed on April 12, 2004.

An investigation into LAS's products and services was performed that involved me, at least one other individual at LAS, and my patent attorneys. The purpose of the investigation was to determine which, if any, of LAS's products and services were prior art to the present application or should otherwise be disclosed to the U.S. Patent & Trademark Office ("PTO"). In the course of the investigation, we determined that PC-NAS had never been disclosed outside of LAS.

During the investigation, however, we did determine that four products/services of LAS should be disclosed to the PTO. These four are: (i) Arabic Name Classifier, (ii) Arabic Name Analyzer, (iii) Consular Lookout And Support System, and (iv) Distributed Name Check. These four products/services are each described and disclosed in another Declaration by me that was filed in the present case on July 13, 2004, as part of an Information Disclosure Statement.

I am aware that the website www.archive.org (the "archive website") has a number of documents purporting to be archives of the LAS website on various dates. I do not know whether or not these documents are accurate. The archive website includes four documents, and only four, that are dated prior to March 25, 1998, which is the priority date of the present application. The dates of the four documents, according to the archive web site, are: 2/1/1997, 7/11/1997, 10/21/1997, and 2/6/1998. The Examiner notes in the 105 requirement that the 10/21/1997 document on the archive website describes a Suite of Tools including NameCheck, NameClassifier, NameRegularizer, Intelligent Search Data Generator, and PhoneticNameKey tools. The Examiner requests disclosure relating to these tools.

During the investigation I reviewed printouts of all four of the www.archive.org documents dated prior to March 25, 1998. During my review, I noticed that the oldest two documents did not contain material describing the Suite of Tools mentioned by the Examiner. That is, the Suite of Tools first appeared in the 10/21/1997 document, and does not appear in either the 2/1/1997 document or the 7/11/1997 document. Accordingly, the Suite of Tools is not in a document having a date one year prior to the priority date. Further, to the best of my knowledge, the Suite of Tools was not disclosed outside of LAS more than one year prior to March 25, 1998.

I hereby declare that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true; and further that these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code and that such willful false statements may jeopardize the validity of the application or any patents issued thereon.

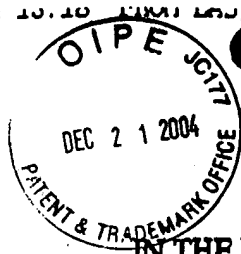
Applicant : John Christian Hermansen et al.
Serial No. : 10/055,178
Filed : January 25, 2002
Page : 3 of 3

Attorney's Docket No.: 16441-012002

Date: Dec 10, 2004

JCHermansen
John C. Hermansen

40256251.doc



Attorney's Docket No.: 16441-012001

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE.

Applicant : John Chrisitan Hermansen et al. Art Unit : 2172
Serial No. : 09/275,766 Examiner : Joon H. Hwang
Filed : March 25, 1999
Title : **SYSTEM AND METHOD FOR ADAPTIVE MUTLI-CULTURAL SEARCHING
AND MATCHING OF PERSONAL NAMES**

Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

DECLARATION OF JOHN CHRISTIAN HERMANSEN

This declaration relates to the following systems: Arabic Name Classifier ("ANC"), Arabic Name Analyzer ("ANA"), Consular Lookout And Support System ("CLASS"), and Distributed Name Check ("DNC").

ANC

To the best of Declarant's recollection, ANC was written as a design document and delivered to a customer no later than the end of 1996.

ANC accepted a romanized input name and a COB associated with the input name, and produced a binary result indicating whether the input name was considered to be Arabic. Specifically, ANC determined a single surname for the input name, and compared that surname against a list of surnames that were known both to be from the COB and to be Arabic. If there was an exact spelling match, then ANC determined that the input name was Arabic and reported this determination to a user. If there was not an exact match, then ANC (i) performed a digram analysis on the input surname to determine the digrams present, (ii) produced an indicator of the similarity between the digram analysis and digram results for Arabic surnames from the COB, (iii) compared the value of the indicator to a threshold value representing confidence in the similarity and, based on this comparison, produced a binary result indicating whether the input name was considered to be Arabic, and (iv) reported the binary result to a user.

ANA

To the best of Declarant's recollection, ANA was written as a design document and delivered to a customer no later than the end of 1996. ANA accepted a romanized input name

Applicant : John Christian Hermansen et al.
Serial No. : 09/275,766
Filed : March 25, 1999
Page : 2 of 4

Attorney's Docket No.: 16441-012001

known to be Arabic, and applied various modification rules to a single surname of the input name. The rules were based on known spelling differences in Arabic surnames, and application of the rules produced a resulting surname which could be different from the surname of the input name. ANA then produced a key representing the resulting surname, and the key could be used to pull names from a database.

CLASS

To the best of Declarant's recollection, no later than the end of 1991 (i) according to the terms of a contract with the United States government, and for compensation, Language Analysis Systems provided a design to the United States government in the United States of America proposing linguistics processing features for CLASS, (ii) according to the terms of a contract between another party (not Language Analysis Systems) and the United States government, and for compensation, CLASS was implemented in software by the other party, with the implementation generally following the proposed design from Language Analysis Systems, and CLASS was provided to the United States government in the United States of America, and (iii) CLASS was operated on a mainframe in the United States of America and accessed by terminals in one or more foreign countries.

CLASS accepted an input name and determined a rank-ordered list of names from a database, where the names in the list were considered to be possible matches for the input name. More specifically, CLASS:

(1) received the input name and various related or corresponding inputs including one or more "compressed name" ("CN") key(s) for corresponding surname(s), a corresponding COB, a corresponding date of birth ("DOB"), and possibly a corresponding state of birth,

(2) identified component elements of the input name (e.g., surname and given name), and identified a first initial of the given name,

(3) identified digrams within each separate component element of the input name ("input name digrams"),

(4) derived a set of names from within a database for comparison to the input name, the set of names being derived based on the input name, the one or more CN keys, the DOB, and the first initial of the given name,

Applicant : John Christian Hermansen et al.
Serial No. : 09/275,766
Filed : March 25, 1999
Page : 3 of 4

Attorney's Docket No.: 16441-012001

(5) identified digrams for the component elements of the names in the set of names ("database name digrams"),

(6) selected a set of weighting rules for producing a score indicating the extent to which two names matched each other, the set of weighting rules being selected based on the COB of the input name,

(7) compared the input name with each name in the set of names, the comparison including comparing the input name digrams to the database name digrams,

(8) generated a metric for each name in the set of names by applying the set of weighting rules during the comparison of the input name with each name in the set of names,

(9) rank-ordered all names in the set of names having a metric greater than a threshold score, the threshold score indicating that the input name matches a particular name from the set of names, and

(10) provided the rank-ordered names to the user.

The set of weighting rules assigned various points to a particular name in the set of names based on a comparison of the particular name and the input name. For example, various points might be assigned depending on whether (i) corresponding element(s) in the particular name and the input name had similar digram results, (ii) the length of one or more elements was the same in the particular name and the input name, (iii) the DOBs of the particular name and the input name were within a predetermined timeframe of each other, (iv) the COB of the input name was the same as the COB associated with the particular name, (v) the elements of the particular name and the input name were in the same order, and (vi) the state of birth was the same for both the particular name and the input name.

DNC

To the best of Declarant's recollection, no later than February 1997 (i) according to the terms of a contract with the United States government and for compensation, DNC was developed by Language Analysis Systems as a computer program and delivered by Language Analysis Systems to the United States government in the United States of America, and (ii) DNC was operated in one or more foreign countries. DNC was similar to CLASS, as described above, except that (i) DNC did not receive or use a key for the surname(s) of the input name, (ii) DNC

Applicant : John Christian Hermansen et al.
Serial No. : 09/275,766
Filed : March 25, 1999
Page : 4 of 4

Attorney's Docket No. 16441-012001

derived the set of names based on the DOB, the COB, and the state of birth (if available), and without reference to a key or the first initial of the given name, and (iii) DNC ran on a personal computer and not on a mainframe, so that when operated in a foreign country DNC only ran on a personal computer in the foreign country and not on a mainframe in the United States of America.

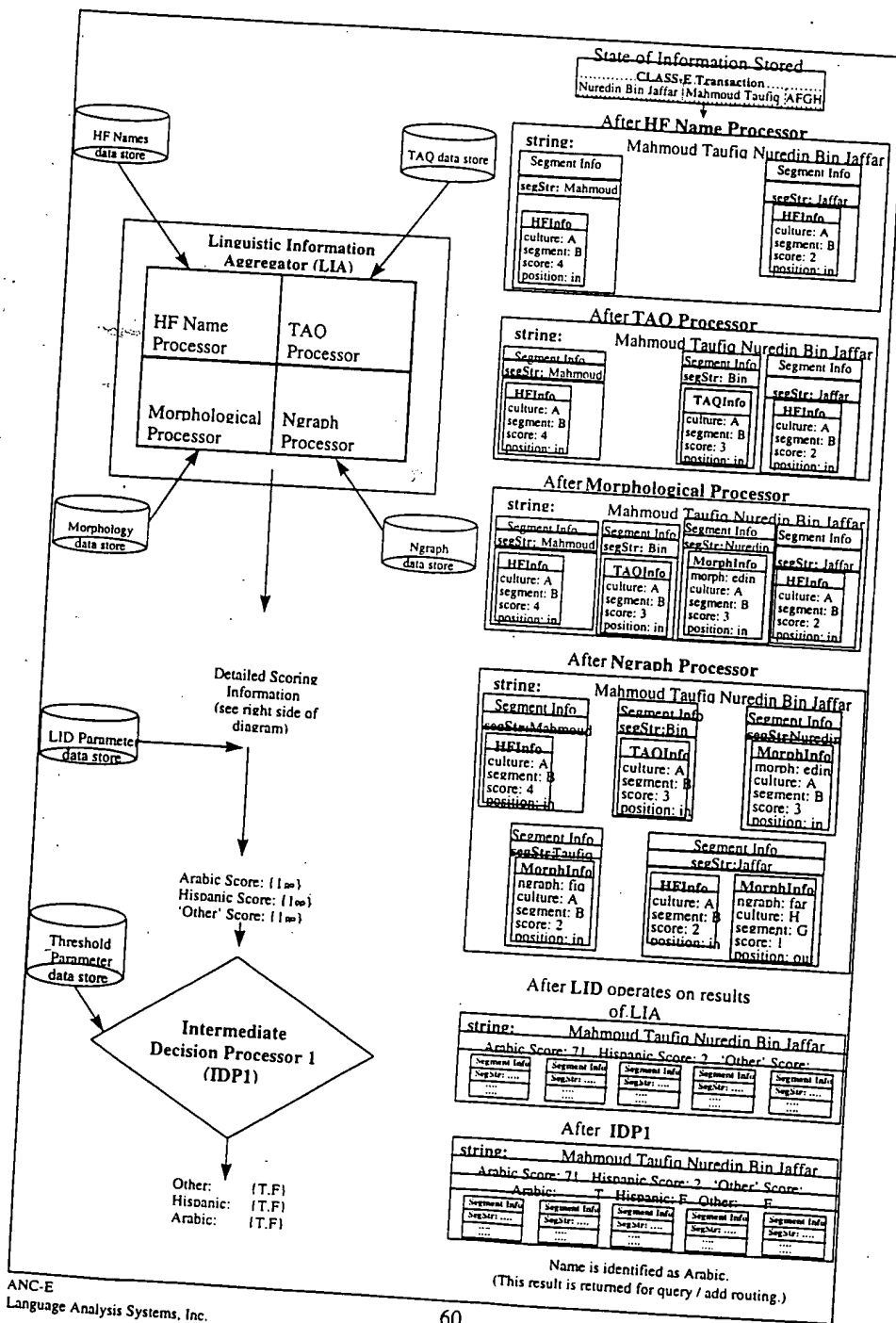
I hereby declare that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true; and further that these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code and that such willful false statements may jeopardize the validity of the application or any patents issued thereon.

Respectfully submitted,

Date:

July 9, 2004

John Christian Hermansen
John Christian Hermansen



SOFTWARE DESIGN DESCRIPTION AUTOMATIC NAME CLASSIFIER FOR CLASS-E (ANC-E)

1. INTRODUCTION

1.1. Project Background

1.1.1. Legacy Consular Lookout And Support System (CLASS) and CLASS-E

The Consular Lookout and Support System (CLASS) performs namechecks of visa and passport applicants in support of the issuance process. Used by United States passport agencies, consulates, and border inspection agencies, CLASS serves as an automated index to manual files. CLASS is a centralized system residing on mainframe computers at the Department of State in Washington, DC. The Bureau of Consular Affairs, Consular Systems Division (CA/EX/CSD) of the Department of State (DOS) has responsibility for development, maintenance, and operation of CLASS.

CLASS was implemented in 1989; since that time, major advancements have occurred in database management systems, large-scale computers and their operating systems, and data telecommunications. In addition, name-matching techniques have also evolved based on the DOS's experience with the system and further linguistic research. This has led DOS in determining the necessity for a newer, more modernized system, CLASS-E (Consular Lookout and Support System-Enhanced).

The CLASS-E modernized version of automated name-matching will incorporate state-of-the-art hardware, data telecommunications, and database management technology to migrate the CLASS application from its Virtual Storage Access Method (VSAM) environment into a DB2 relational database system. In addition to providing virtually uninterrupted access to the lookout databases 24 hours a day, 7 days a week to the VO, PPT, overseas posts and support users, this enhanced system will position CLASS-E to incorporate advanced culturally-sensitive namecheck methods.

1.1.2. Culturally Sensitive Name Searching in CLASS-E

Personal naming systems vary widely from culture to culture. That is, names from around the world do not necessarily fit cleanly into the

Anglophone name model. Several of the manifestations of these differences are

- Anglicization of Non-English sound patterns (Miladevic written as Miladevich)
- Variant romanization schemes (Arabic Waseem ~ Ouassime, Shareef ~ Cherife; Chinese Xia ~ Hsia ~ Sya)
- Dialectal variants (Arabic Abu Bakir [Egyptian] ~ Boubker [Moroccan]; Chinese Wu [Mandarin] ~ Ng [Cantonese, Fukien])
- Variant roman spelling conventions (French silent letters, German sch for English sh)

When dealing with Arabic and Chinese names and those of other languages that do not use the Roman alphabet, for example, one quickly discovers one major source of name variation lies in how names are transliterated into roman characters from the original scripts. For both Arabic and Chinese, there are numerous competing transliteration standards, as well as less formal traditions. Xia, Hsia, and Sya, for example, are all romanized variants of the same Chinese name. Kassim, Qasim, Casem, Kacem and Asim are romanized variants of the same Arabic name. In Arabic, name variation often goes beyond the phonetic level. Analyzable elements such as "Abu" show up in many different forms, depending on dialect (e.g., Abu Bakir ~ Boubker). In Chinese, multiple traditions of transliteration are one of the sources of name variation; dialect issues also abound (e.g., Wu ~ Ng). Hispanic names, which make up the largest portion of the data base, place information value on name parts in a manner that is not consistent with Anglophone naming conventions. Exploitation of this culturally-specific information in the name search process leads to improved precision, recall, and overall system performance.

1.1.3. Automatic Name Classifier-E (ANC-E) in CLASS-E

The need for automatic name classification has become a necessary first step in the process of applying linguistic knowledge to solve the problems associated with name searching in large multicultural databases. In this environment, name classification serves as a means of routing queries to the proper language- and culture-specific algorithms. Currently, Legacy CLASS supports a single module, called ANI, which begins to address this need by returning a Boolean value indicating whether a name is or is not Arabic. If a name qualifies as Arab, it is subject to processing by an initial implementation of the Arabic algorithm designed by LAS for the State Department. Currently the expanding needs of the State Department are being addressed in the development of a second culture-specific algorithm which will handle Hispanic names. The addition of a Hispanic algorithm to CLASS's

functionality requires the addition of a method for identifying Hispanic names in a manner similar that of ANI.

At this juncture, it is reasonable to turn enhancement efforts towards the development of a single, integrated, expandable algorithm for name classification which will address the need for classifying Arabic and Hispanic names, and which will anticipate the imminent addition of other languages. The integrated automatic name classification algorithm will represent a significant improvement over the existing ANI algorithm in that it will incorporate more linguistic knowledge, it will allow for future expansion with minimal coding effort, and it will allow information about a record's country of birth (COB) to contribute to the query routing decision. Figure 1-1 displays the integration of ANC-E within the CLASS-E system.

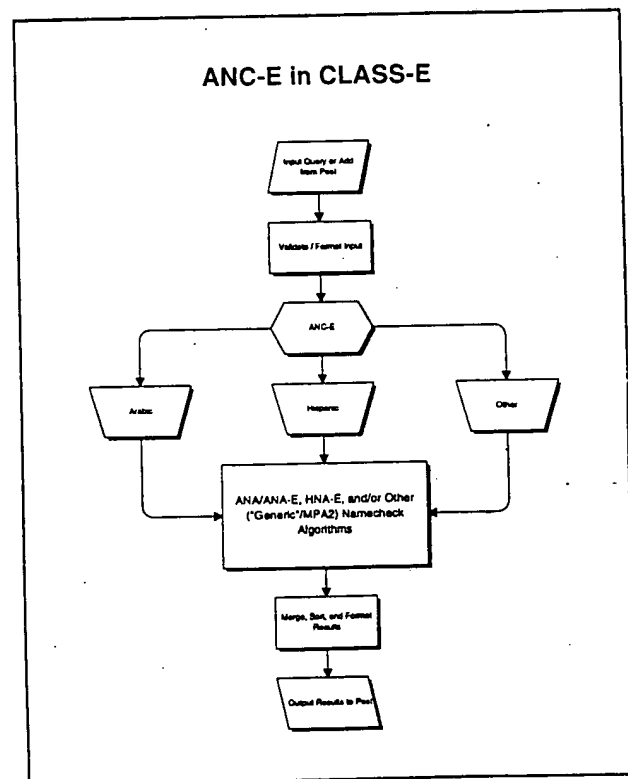


Figure 1-1

1.2. Scope

This document describes the linguistic motivation, requirements, and high level design for an Automatic Name Classifier (ANC) which will automatically determine whether a name qualifies as Hispanic or Arabic. The document's purpose is to provide information about the proposed design in order to facilitate the analysis and planning necessary to prepare for eventual implementation.

Intended to serve as the module that will provide for the integration of the enhanced Arabic Name Search Algorithm for CLASS-E (ANA-E) and the Hispanic Name Search Algorithm (HNA-E) into the overall CLASS-E architecture, the Automatic Name Classifier for CLASS-E (ANC-E) will provide the capability to automatically determine whether an input name is Arabic, Hispanic, or neither. In this system, names may be qualified as Arabic or Hispanic by virtue of passing one of two thresholds, or, conversely, may be disqualified as Arabic or Hispanic by virtue of having many characteristics of 'Other' types of names. The ANC-E system has been designed with an open architecture intended to facilitate the inclusion of additional cultures in the event that CLASS-E adds other culture-specific search algorithms in the future. Furthermore, since the ANC-E is data-driven, it is possible to tune its level of sensitivity for each individual culture being identified.

In CLASS-E the concept of the Legacy CLASS Multi-Pipe Architecture will be carried forward to include a distinct Arabic processing algorithm and a distinct Hispanic processing algorithm as well as perhaps others in the future. The type of processing to which an input name will be submitted will be a business decision of CA/EX/CSD and may to some degree be dependent on the impact that multiple processing of an input name would have on the performance of the system. It is likely that input names that are classified by the Advanced Name Classifier for CLASS-E (ANC-E) will be submitted to multiple of the following processors: the generic CLASS-E generic processing algorithm, the DOB processing algorithm, the ANA-E algorithm, and the HNA-E algorithm. The ANC-E will provide a determination as to which culture or cultures a name belongs; what use is made of this determination is a business decision of CA/EX/CSD. This decision will affect the design of the interface between the ANC-E and the rest of the CLASS-E system.

1.3. Definitions and Acronyms

1.3.1. Definitions

Affix

A name *particle* which is neither a *title* nor a *qualifier*. Affixes in the ANC-E are defined as being delineated by white space; for example,

Digraph
Field

'de' in 'Tirso de Molina'. Note that, contrary to normal usage within linguistics, affixes are in contrast to (bound) *morphemes*, which are not delineated by white space.

A two character *n-gram*.

A data entry mechanism which allows the user to input a fixed number of characters. The fields typically referred to in the CLASS environment are the Given Name Field and the Surname Field.

Given Name

Note that it is important to distinguish between *given name* and *surname* data entry Fields and *given name* and *surname* data elements, since data elements do not always occur in the proper field.

The portion of a *name* which uniquely identifies an individual member of a family, as opposed to *surname*. Given Names may include one or more segments; for example, 'Mary Jane' in 'Mary Jane Cassoway'.

Infix

A substring occurring the middle of a name segment, but not at the edges. Both *n-grams* and *morphemes* may be infixes.

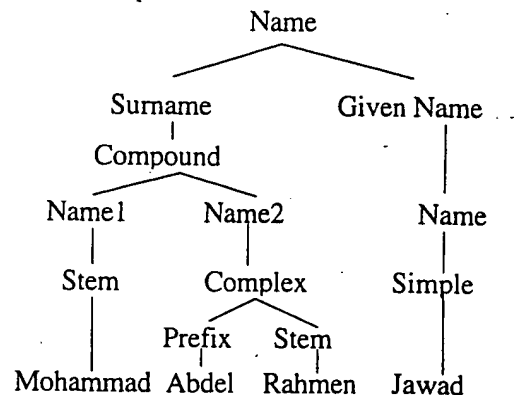
Morpheme

(here, *bound* morpheme) A meaningful, variable length substring of a name segment. Morphemes may occur as *prefixes*, *infixes* or *suffixes*. Examples: '-ovitch' in 'Berkovitch'. Note that morphemes contrast with *affixes*.

Morphology
Name

Referring to *morphemes*.

The general term referring to the entire collection of *segments* which refer to a single person. A name may include one or more *given names*, one or more *surnames* and zero or more *particles*. For the purposes of ANC-E, a Name is considered to consist only of alphabetic characters and white space. The diagram below illustrates the relation of name parts to one another:



N-Gram

A variable length sequence of characters which serves as a useful

* Note that these terms have a slightly modified or restricted definition within the context of ANC-E.

	indicator of linguistic affinity, but which is not associated with a meaning. N-Grams may be considered to be indicators of the sound or spelling patterns of a language; for example, -ez is a Hispanic N-Gram.
Particle	A functional name element delineated by white space. <i>Titles, affixes</i> and <i>qualifiers</i> are the three kinds of particles identified in the ANC-E algorithm.
Prefix	A substring (<i>N-Gram</i> or <i>morpheme</i>) or a <i>particle</i> (<i>affix</i>) occurring at the beginning of a name segment.
Qualifier	A meaningful <i>particle</i> which represents a kinship relation or earned social status; for example, Jr. or Ph.D. Qualifiers typically occur at the end of a name field.
Segment	Any element within a name which is delineated by white space.
Suffix	A substring (<i>N-Gram</i> or <i>morpheme</i>) or a <i>particle</i> (<i>affix</i>) occurring at the end of a name segment.
Surname	The portion of a <i>name</i> which may indicate family membership, as opposed to <i>given name</i> . Surnames may include one or more segments and zero or more <i>particles</i> ; for example, 'Fernandez de la Puente' in 'Hector Fernandez de la Puente'.
Syntax	The rules governing the order of name elements.
Title	A meaningful <i>particle</i> which represents a term of address and which typically occurs at the beginning of a name field. Examples: Dr. or Sir. Titles may be indicative of social position.
Trigraph	A three character <i>n-gram</i> .
Variant	An alternate spelling of a name <i>segment</i> ; for example, Mohammad and Muhamed are variants of one another. Variants may be predictable, as in this example, or unpredictable, as evidenced by typographical or other data entry errors.

1.3.2. Acronyms

ANA	Legacy Arabic Namecheck Algorithm
ANA-E	Arabic Namecheck Algorithm for CLASS-E
ANC-E	Automatic Name Classifier for CLASS-E
ANI	Arabic Name Identification (of Legacy CLASS ANA)
ANR	Arabic Name Regularization (of Legacy CLASS ANA)
AOR	Application Owning Region
ARTP	Acceptance/Regression Test Plan
ARTR	Acceptance/Regression Test Report
BIMC	Beltsville Information Management Center
C/CE	CLASS to CLASS-E
CA	Bureau of Consular Affairs

CA/EX/CSD	Consular Affairs, Consular Systems Division
CAX	Consular Affairs Experimental (Development)
CCB	Configuration Control Board
CCR	Configuration Change Request
CDD	Critical Design Document
CDR	Critical Design Review
CE	CLASS-Enhanced
CICS	Customer Information Control System
CLASS	Consular Lookout and Support System
CLASS-E	Consular Lookout and Support System-Enhanced
CM	Configuration Management
CMOS	Complementary Metal Oxide Semiconductor
COB	Country of Birth
COR	Contracting Office Representative
CSD	Computer Systems Division
DBMS	Database Management System
DB2	IBM's relational database
DIA	Digraph Information Aggregator (of ANC-E)
DNC	Distributed Namecheck
DOB	Date of Birth
DOS	Department of State
FRR	Functional Requirements Review
FRS	Functional Requirements Specification
HNA-E	Hispanic Namecheck Algorithm for CLASS-E
IBIS	Interagency Border Inspection System
IDP1/IDP2	Intermediate Decision Processor 1 / 2 (of ANC-E)
IP	Installation Plan
IVV	Independent Verification and Validation
LIA	Linguistic Information Aggregator (of ANC-E)
LID	Linguistically Informed Decision Processor (of ANC-E)
LQA	Linguistic Quality Assurance
LQAR	Linguistic Quality Assurance Report
LSP	Linguistic Support Plan
LTF	Linguistic Trace Facility
NC	Namecheck
PC	Production Control
PMP	Project Management Plan
PPP	Post Phase-In Plan

PPT	Passport Office
PTS	Parallel Transaction Server
QA	Quality Assurance
QMF	Query Management Facility
QRP	Query Routing Processor
SA-1	State Annex-1
SESAP	Software Engineering Standards and Procedures
TAQ	Title, Affix Qualifier
TIR	Test Incident Report
TOR	Terminal Owning Region
TRR	Test Readiness Review
VO	Visa Office
VSAM	Virtual Storage Access Method

2. References

- 2.1. CLASS-E Project Management Plan (PMP)
- 2.2. CLASS-E Functional Requirements Specification (FRS)
 - 2.2.1. Note: the CLASS-E FRS has not yet been finalized.

3. Decomposition Description

3.1. Module Decomposition

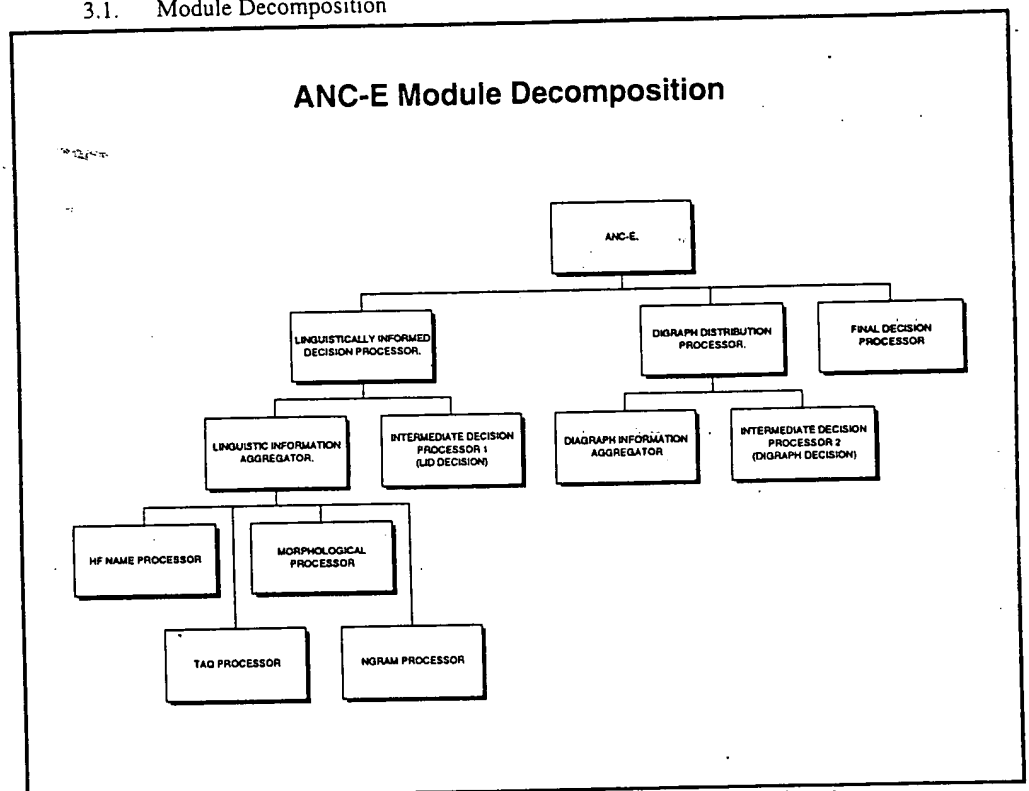


Figure 3-1

3.1.1. Automatic Name Classifier for CLASS-E (ANC-E) Module Decomposition

3.1.1.1. Identification

This program is referred to as the Automatic Name Classifier for CLASS-E (ANC-E).

3.1.1.2. Type

ANC-E is a program that is part of the larger CLASS-E system. It can be viewed as a "shell" program in that it is to

serve as a layer surrounding all of the culturally-specific name search algorithms implemented in CLASS-E.

3.1.1.3. Purpose

- 3.1.1.3.1. The need for automatic name classification is a necessary first step in the process of applying linguistic knowledge to solve the problems associated with name searching in large multicultural databases.
- 3.1.1.3.2. In the CLASS-E environment, name classification serves as a means of routing queries to the proper language- and culture-specific algorithms.
- 3.1.1.3.3. In addition to the rudimentary identification of Arabic names currently implemented in ANI, the addition of a Hispanic name search algorithm to CLASS-E's functionality requires the addition of a method for identifying Hispanic names.
- 3.1.1.3.4. ANC-E is a single, integrated algorithm for name classification which will address the need for classifying Arabic and Hispanic names, and which will anticipate the possible addition of other languages.
- 3.1.1.3.5. This integrated automatic name classification algorithm will represent a significant improvement over the existing ANI algorithm in that it will incorporate more linguistic knowledge, and will allow information about a record's country of birth (COB) to contribute to the query routing decision.

3.1.1.4. Function

- 3.1.1.4.1. The ANC-E will take as input a surname, given name, and COB in standard CLASS-E format.
 - 3.1.1.4.1.1. There are two options with respect to the methodology for handling an input name and gathering the aggregate data

that will lead to the determination of cultural affinity for that name.

3.1.1.4.1.1.1. If ANC-E is to be implemented in an object-oriented environment, an object can be created which will contain all of the accumulated information to be used in the determination of cultural affinity. This object travels through the ANC-E system, thus allowing access to the accumulated information at any time. If ANC-E is integrated with the culturally-sensitive name search algorithms in CLASS-E, this option has the advantage that the all of the attendant linguistic information produced by ANC-E processing could be passed, along with the name, to the culturally-sensitive namecheck algorithm for further processing. That is, certain common linguistic processing would need to be performed only one time for the entire namecheck process, rather than once for each specific name search algorithm invoked.

3.1.1.4.1.1.2. If ANC-E is to be implemented in a non-object-oriented environment, ANC-E will process the name and COB as separate string values, and will output either a single cultural affinity indicator (e.g. Arabic, Hispanic, or Other) or three Boolean values, one for each

culture under consideration, depending on the business decision made by CA/EX/CSD. If this option is chosen, linguistic processing information and scoring internal to ANC-E will not be available to outside processes.

3.1.1.4.2. The ANC-E will provide a determination as to which culture or cultures a name belongs.

3.1.1.4.3. The use that is made of the cultural affinity determinations made by ANC-E is a business decision of CA/EX/CSD (i.e. whether to allow a name to be processed by more than one namecheck algorithm, and whether ANC-E shall return more than one possible cultural affinity for a given input name). This decision will affect the design of the interface between the ANC-E and the rest of the CLASS-E system.

3.1.1.5. Subordinates

The following processes are subordinate to the main ANC-E program:

- The Linguistically Informed Decision Processor (LID)
- The Digraph Distribution Processor
- The Final Decision Processor.

3.1.2. Linguistically Informed Decision (LID) Module Decomposition

3.1.2.1. Identification

This module is referred to as the Linguistically Informed Decision Processor (LID).

3.1.2.2. Type

The LID is a module which contains two subordinate modules. The first subordinate module performs linguistic analysis, gathering linguistic information and scoring for the input name. The second subordinate module makes decisions as to the cultural affinity of the name, based on the scoring information gathered by the first module.

3.1.2.3. Purpose

- 3.1.2.3.1. The LID exists to provide a linguistically well-founded decision as to the cultural affinity of the input name.
- 3.1.2.3.2. As the first phase of processing, the LID addresses performance requirements by basing this decision on multiple readily observable linguistic factors, thus obviating the need for processing by the more intensive statistical digraph model and for reliance on name-external factors, such as Country of Birth (COB).
- 3.1.2.3.3. Furthermore, the LID provides a more linguistically-rich context in which to determine the cultural affinity of the input name than does its purely digraph-distribution-based predecessor, ANI. Thus ANC-E is better able to identify names that are Hispanic or Arabic and to eliminate those that are not. Linguistic Indicators provide a rich source of information about the cultural affinity of a name. The LID processor will serve as a means of assuring that names which are strongly Arabic or Hispanic are qualified and, conversely, that names which have strong characteristics of some other culture are disqualified. Names which qualify as Hispanic, Arabic or 'Other' will not be submitted to the Digraph Analysis function.

3.1.2.4. Function

- 3.1.2.4.1. All linguistic indicator processing will take place before digraph analysis and will constitute a linguistically informed decision (LID) mechanism.
- 3.1.2.4.2. The LID accumulates and weighs factors from multiple knowledge sources in order to determine whether there is a sufficient amount of evidence to identify the input name as being Hispanic or Arabic, or, conversely, if there is enough

evidence to discount the possibility that the input name is either Hispanic or Arabic.

- 3.1.2.4.3. The LID will assign points to a name based on a weighted tabulation of scores from the following data sources:
- High Frequency name data
 - TAQ data
 - Morphological data
 - Ngram data
- 3.1.2.4.4. The function of the LID is to determine a score for each cultural affinity being classified, and a score for 'Other'. For each culture, a name must get a score which passes its corresponding LID Threshold in order to be labeled as Arabic, Hispanic or "Other".
- 3.1.2.4.5. Each of the four types of linguistic indicator (listed in 3.1.2.4.3) will be associated with a set of four parameters, indicating the weight that a LID element is to be given.
- 3.1.2.4.6. The score for each language group will be calculated as a summation of the combination of the applicable factor times the score for each indicator found in the name string. Scoring details are included in the decomposition descriptions of the respective modules. (See sections 3.1.3 - 3.1.8.)
- 3.1.2.4.7. After all of the agents have processed the input name, the LID combines the detailed scoring information returned by the LIA to produce a LID score for Hispanic, Arabic, and for Other.
- 3.1.2.4.8. The LID passes the LID score to the Intermediate Decision Processor 1 for comparison to LID thresholds for cultures under consideration.
- 3.1.2.4.9. There are two alternatives for the output of the processing of the input name performed by the LID: an object containing linguistic processing information and scores or three Boolean values indicating whether the name has passed the LID

thresholds for Arabic, Hispanic, or Other. For more information, see 3.1.1.4.1.1.

3.1.2.4.10. If the LID identifies a name as Hispanic, Arabic, *or* Other (or any combination thereof), no further processing is required.

3.1.2.4.11. For a detailed example of LID processing, see the figures in Appendix A.

3.1.2.5. Subordinates

The following processes are subordinate to the LID:

- The Linguistic Information Aggregator (LIA)
- Intermediate Decision Processor 1 (LID Decision).

3.1.3. Linguistic Information Aggregator (LIA) Module Decomposition

3.1.3.1. Identification

This module is referred to as the Linguistic Information Aggregator (LIA).

3.1.3.2. Type

LIA is a module which contains four subordinate functions (agents) all of which contribute to the final decision or decisions made by the LID as to the cultural affinity of the input name. Thus, conceptually, LIA and the LID can be viewed as parts of a blackboard (voting) system, an expert system, or as parts of a system with multiple intelligent agents.

3.1.3.3. Purpose

3.1.3.3.1. The LIA exists to enable the linguistic decision made by the LID. The LIA controls the flow of information from the four linguistic agents subordinate to it.

3.1.3.3.2. If the implementation choices accompanying the object-oriented description of ANC-E are chosen (see 3.1.1.4.1.1.1), LIA could help performance by allowing certain linguistic processing to occur only once for each name check, rather than once for each algorithm invoked. (Note: In Legacy CLASS each algorithm is referred to as a separate 'pipe'.)

3.1.3.4. Function

3.1.3.4.1. LIA accumulates linguistic information factors from multiple knowledge sources for each culture under consideration (i.e. currently Hispanic, Arabic, and Other).

3.1.3.4.2. In cases of conflict, the order of precedence for identifying items within an input name is TAQ particle, Morpheme, Ngram.

3.1.3.4.2.1. If a string of letters is identified as a TAQ particle for a particular culture, a substring of that same string (including the entire string itself) cannot also be identified as a Morpheme or an Ngram for that same culture.

3.1.3.4.2.2. If a string is identified as a Morpheme for a particular culture, the characters that make up that Morpheme cannot also be considered as part of an Ngram for that culture.

3.1.3.4.2.3. HF Names from a given culture can contain Morphemes and / or Ngrams for that same culture; however, the precedence rules in sections 3.1.3.4.2.1 and 3.1.3.4.2.2 apply.

3.1.3.4.3. As the subordinate functions (agents) process the input name, detailed scoring information is collected by LIA, and weighted according to its information value as indicated in the LID Parameter data store.

3.1.3.4.4. After all of the agents have provided their input, the LIA returns this detailed scoring information to the LID.

3.1.3.4.5. For a detailed example of aggregation of information by LIA, see the figures in Appendix A.

3.1.3.5. Subordinates

The following processes are subordinate to the LIA:

- The High Frequency (HF) Name Processor
- The Title, Affix, Qualifier (TAQ) Processor
- The Morphological Processor
- The Ngram Processor.

3.1.4. High Frequency (HF) Name Processor Module Decomposition

3.1.4.1. Identification

This function is referred to as HF Name Processor.

3.1.4.2. Type

The HF Name Processor is a function which is invoked by the Linguistic Information Aggregator (LIA).

3.1.4.3. Purpose

Certain given names and surnames occur much more frequently in some cultures than in others. The name "Mohammed", for example occurs frequently in Arabic names. The surname "Rodriguez" lends support to the possibility that the name in question is Hispanic. The name "Nganga" in any position suggests that the name might not be either Arabic or Hispanic. The HF Name Processor exists to take advantage of the information available in high frequency names in the cultural identification of the name.

3.1.4.4. Function

- 3.1.4.4.1. For each name segment present in the input name, the HF Name Processor determines whether that name is present in the HF Name data store.
- 3.1.4.4.2. If the name is present in the HF name data store, the HF Name Processor retrieves and records the culture, name field (given name or surname), and score associated with that name from the data store.
- 3.1.4.4.3. Also recorded for each HF name found is whether it was found in position or out of position. For example, since "Rodriguez" is listed as a surname in the HF Names data store, if it is found in the GN field in the input name, it

is reported as a surname considered to be out of position.

3.1.4.4.4. The HF Name Processor tracks scoring information for each HF name found, and returns this detailed scoring information to LIA.

3.1.4.5. Subordinates
None.

3.1.5. Title, Affix, Qualifier (TAQ) Processor Module Decomposition

3.1.5.1. Identification

This function is referred to as the TAQ Processor.

3.1.5.2. Type

The TAQ Processor is a function which is invoked by the Linguistic Information Aggregator (LIA).

3.1.5.3. Purpose

As noted in section 1.3.1, name fields have a syntactic structure which may be simple, compound, complex, or compound-complex. Name fields which are complex or compound-complex contain particles: titles, affixes, or qualifiers. These particles can be used to further narrow the range of possibilities for the cultural affinity of the input name. The TAQ Processor exists to make use of the information available in particles.

3.1.5.4. Function

3.1.5.4.1. For each segment present in the input name, the TAQ Processor determines whether that segment is a particle present in the TAQ data store.

3.1.5.4.2. If the segment is present in the TAQ data store, the TAQ Processor retrieves and records the culture, name field (given name or surname), and score associated with that TAQ particle from the data store.

3.1.5.4.3. Also recorded for each TAQ particle found is whether it was found in position or out of position. (See example in section 3.1.4.4.3.)

3.1.5.4.4. The TAQ Processor tracks scoring information for each HF TAQ particle found, and returns this detailed scoring information to LIA.

3.1.5.5. Subordinates

None.

3.1.6. Morphological Processor Module Decomposition

3.1.6.1. Identification

This function is referred to as Morphological Processor.

3.1.6.2. Type

The Morphological Processor is a function which is invoked by the Linguistic Information Aggregator (LIA).

3.1.6.3. Purpose

As noted and defined in section 1.3.1, morphological elements, such as -ovich, can play a large part in determining the cultural affinity of an input name. The Morphological Processor exists to take advantage of this information in the name classification process.

3.1.6.4. Function

3.1.6.4.1. For each Morpheme present in the Morphology data store, the Morphological Processor determines whether that Morpheme is present in the input name.

3.1.6.4.1.1. Note that the above processing differs from that in the HF Name Processor (3.1.4.4) and the TAQ Processor (3.1.5.4). Since the Morphology data store contains only bound Morphemes, that is Morphemes not surrounded by white space, it is not possible to locate them based on name segments, which are surrounded by white space. Rather, it is necessary to determine if any of the items listed in the

Morphology data store is a substring of any of the name segments present in the input name, within certain constraints. For more detailed information on identifying Morphemes in the input name, see sections 3.2.4 (Morphological Data Store Data Decomposition) and 3.1.6 (Morphological Processor Module Decomposition).

3.1.6.4.2. For each Morpheme found in the input name, the Morphological Processor retrieves and records the morpheme found, the culture, name field (given name or surname), and score associated with that Morpheme from the data store.

3.1.6.4.3. Also recorded for each Morpheme found is whether it was found in position or out of position. (See example in section 3.1.4.4.3.)

3.1.6.4.4. The Morphological Processor tracks scoring information for each Morpheme found, and returns this detailed scoring information to LIA.

3.1.6.5. Subordinates
None.

3.1.7. Ngram Processor Module Decomposition

3.1.7.1. Identification

This function is referred to as the Ngram Processor.

3.1.7.2. Type

The Ngram Processor is a function which is invoked by the Linguistic Information Aggregator (LIA).

3.1.7.3. Purpose

As described in section 1.3.1, Ngrams are strings of letters that occur with statistical significance in names with a given cultural affinity. The Ngram Processor exists to take advantage of this statistical phenomenon in the name typing process.

3.1.7.4. Function

3.1.7.4.1. For each Ngram present in the Ngram data store, the Ngram Processor determines whether that Ngram is present in the input name.

3.1.7.4.1.1. Note that the above processing is similar to that in the Morphological Processor. (See section 3.1.6.4, and especially section 3.1.6.4.1.1 for a detailed note.)

3.1.7.4.2. For each Ngram found in the input name, the Ngram Processor retrieves and records the Ngram found, the culture, name field (given name or surname), and score associated with that Ngram from the data store.

3.1.7.4.3. Also recorded for each Ngram found is whether it was found in position or out of position. (See example in section 3.1.4.4.3.)

3.1.7.4.4. The Ngram Processor tracks scoring information for each Ngram found, and returns this detailed scoring information to LIA.

3.1.7.5. Subordinates

None.

3.1.8. Intermediate Decision Processor 1 (LID Decision) Module Decomposition

3.1.8.1. Identification

This module is referred to as Intermediate Decision Processor 1 (IDP1).

3.1.8.2. Type

IDP1 is a function which is invoked directly by the Linguistically Informed Decision Processor (LID).

3.1.8.3. Purpose

IDP1 is the decision-making function of the LID. It determines whether enough linguistic information has been gathered from the various intelligent agents by LIA to

confidently determine that the input name belongs to one of the cultures being identified (currently Arabic, Hispanic, and Other).

3.1.8.4. Function

- 3.1.8.4.1. IDP1 accepts as input one aggregate LID score for each culture being identified as well as an aggregate LID score for Other.
- 3.1.8.4.2. For each LID score, IDP1 compares that score to the LID threshold for the appropriate culture (or Other).
- 3.1.8.4.3. If the LID score is greater than or equal to the appropriate LID threshold, IDP1 returns a value of True for the culture in question. If the LID score is less than the LID threshold for the culture in question, IDP1 returns a value of False for the culture in question.
 - 3.1.8.4.3.1. A True value indicates to the LID that enough evidence has been accumulated by LIA to confidently identify the name as belonging to the culture in question.
 - 3.1.8.4.3.2. A False value indicates to the LID that not enough evidence has been accumulated by LIA to confidently identify the name as belonging to the culture in question.
 - 3.1.8.4.3.3. A value of True can be returned for more than one cultural affinity.
 - 3.1.8.4.3.4. A value of False may be returned for all cultural affinities.
- 3.1.8.4.4. Alternatively, IDP1 could return a value for each culture equal to the LID score minus the LID threshold for that culture.
 - 3.1.8.4.4.1. Given the alternative above, the LID would interpret negative scores as

False values and nonnegative scores as True values.

3.1.8.4.4.2. The utility of this alternative is that if an object-oriented implementation is chosen, the values calculated by IDP1 could be incorporated into the object mentioned in section 3.1.1.4.1.1.1, and would be available as part of the information that the name object "knows" about itself for use in later processing.

3.1.8.4.5. If a return value of True for any culture (or for "Other") is obtained from IDP1, no further processing is required.

3.1.8.5. Subordinates
None.

3.1.9. Digraph Distribution Processor Module Decomposition

3.1.9.1. Identification

This module is referred to as the Digraph Distribution Processor.

3.1.9.2. Type

The Digraph Distribution Processor is a module which has two subordinate functions.

3.1.9.3. Purpose

The Arabic Name Identification (ANI) subprogram currently in use in Legacy CLASS is based purely on a model of digraph distribution in Arabic names. Digraph distribution information has proved useful in determining the cultural affinity of names. Based on a statistical model generated from digraph distribution statistics and initial and final trigraph statistics, the Digraph Distribution Processor lends additional information to the attempt to identify the provenance of the input name.

3.1.9.4. Function

3.1.9.4.1. The Digraph Distribution Processor takes as input the surname from the name input to

ANC-E. This portion of ANC-E operates only on surname data.

- 3.1.9.4.2. The Digraph Distribution Processor is invoked only when the LID has not been successful in assigning any cultural affinity to the input name. (See sections 3.1.2.4.10 and 3.1.8.4.5.)
- 3.1.9.4.3. The Digraph Distribution Processor calculates scores based on digraph distribution statistics for each culture in order to determine whether there is a sufficient amount of evidence to identify the input name as being Hispanic or Arabic. Note that there is no Digraph Distribution Score computed for Other.
- 3.1.9.4.4. The Digraph Distribution Parameters data store contains a Digraph Skew Factor for each cultural affinity.
- 3.1.9.4.5. The Total Digraph Distribution Score for the input name is equal to the Raw Digraph Distribution Score returned by the DIA plus the value of the Digraph Skew Factor for the appropriate culture.
- 3.1.9.4.6. The Digraph Distribution Processor passes the Total Digraph Distribution score for each culture to the Intermediate Decision Processor 2 for comparison to Digraph thresholds for cultures under consideration.
- 3.1.9.4.7. There are two alternatives for the output of the processing of the input name performed by the Digraph Distribution Processor: an object containing a Digraph Distribution Score for each culture, or two Boolean values indicating whether the name has passed the Digraph thresholds for Arabic or Hispanic. For more information, see 3.1.1.4.1.1.
- 3.1.9.4.8. If the Digraph Distribution Processor identifies a name as Hispanic, Arabic, or both no further processing is required.

3.1.9.5. Subordinates

The following processes are subordinate to the Digraph Distribution Processor:

- The Digraph Information Aggregator (DIA)
- Intermediate Decision Processor 2 (Digraph Decision).

3.1.10. Digraph Information Aggregator (DIA) Module Decomposition

3.1.10.1. Identification

This module is referred to as the Digraph Information Aggregator (DIA).

3.1.10.2. Type

The DIA is a process invoked by the Digraph Distribution Processor. The DIA operates only on surname segments consisting solely of alphabetic characters.

3.1.10.3. Purpose

The DIA gathers the information necessary for the Digraph Distribution Processor to determine whether there is sufficient information to identify the input name as Hispanic or Arabic.

3.1.10.4. Function

3.1.10.4.1. For purposes of DIA processing, a surname segment is defined as any string of characters delimited by white space.

3.1.10.4.1.1. Given a surname containing more than one part as input, the name is segmented (based on white space). Each part of multi-part surnames is processed separately, and the scores are combined in the manner described below.

3.1.10.4.2. DIA will calculate a score for each surname segment by totaling the scores for all digraphs within the surname segment.

3.1.10.4.2.1. The set of digraphs for a surname consists of all possible substrings of

two contiguous characters in the surname.

3.1.10.4.2.2. Word-boundaries are considered characters, so the additional digraphs "*word-boundary+first-letter*" and "*last-letter+word-boundary*" are included in the set of digraphs for each name.

3.1.10.4.2.3. In general, a surname segment of length n contains $(n+1)$ digraphs, ordered from leftmost to rightmost.

3.1.10.4.3. Each digraph in the surname segment is looked up in a table containing scores for all possible digraphs for all cultural affinities being scored. DIA maintains a cumulative total of all digraph scores assigned to a surname segment.

3.1.10.4.4. Likewise, scores are assigned for the initial and final trigraphs of each name segment.

3.1.10.4.5. The initial and final trigraph scores are added to the cumulative score for that segment. A score is thus calculated for each segment of the surname.

3.1.10.4.6. The Raw Digraph Distribution Score for the input name is equal to the sum of all individual surname segment scores thus calculated.

3.1.10.5. Subordinates

None.

3.1.11. Intermediate Decision Processor 2 (Digraph Decision) Module Decomposition

3.1.11.1. Identification

This module is referred to as Intermediate Decision Processor 2 (IDP2).

3.1.11.2. Type

IDP2 is a function which is invoked directly by the Digraph Distribution Processor.

3.1.11.3. Purpose

IDP2 is the decision-making function of the Digraph Distribution Processor. It determines whether enough digraph distribution information is present to confidently determine that the input name belongs to one of the cultures being identified (currently Arabic or Hispanic).

3.1.11.4. Function

3.1.11.4.1. IDP2 accepts as input one Digraph Distribution Score for each culture being identified.

3.1.11.4.2. For each Digraph Distribution Score, IDP2 compares that score to the Digraph threshold for the appropriate culture.

3.1.11.4.3. If the Digraph Distribution Score is greater than or equal to the appropriate Digraph threshold, IDP2 returns a value of True for the culture in question. If the Digraph Distribution Score is less than the Digraph threshold for the culture in question, IDP2 returns a value of False for the culture in question.

3.1.11.4.3.1. A True value indicates to the Digraph Distribution Processor that digraph distribution information is conclusive enough to confidently identify the name as belonging to the culture in question.

3.1.11.4.3.2. A False value indicates to the Digraph Distribution Processor that digraph distribution information is not conclusive enough to confidently identify the name as belonging to the culture in question.

3.1.11.4.3.3. A value of True can be returned for more than one cultural affinity.

3.1.11.4.3.4. A value of False may be returned for all cultural affinities.

3.1.11.4.4. Alternatively, IDP2 could return a value for each culture equal to the Digraph Distribution

Score minus the Digraph threshold for that culture.

3.1.11.4.4.1. Given the alternative above, the Digraph Distribution Processor would interpret negative scores as False values and nonnegative scores as True values.

3.1.11.4.4.2. The utility of this alternative is that if an object-oriented implementation is chosen, the values calculated by IDP2 could be incorporated into the object mentioned in section 3.1.1.4.1.1.1, and would be available as part of the information that the name object "knows" about itself for use in later processing.

3.1.11.5. Subordinates

None.

3.1.12. Final Decision Processor Module Decomposition

3.1.12.1. Identification

This module is referred to as the Final Decision Processor.

3.1.12.2. Type

The Final Decision Processor is a module invoked directly by the ANC-E main program.

3.1.12.3. Purpose

3.1.12.3.1. Although the LID and the Digraph Distribution Processor are each powerful methods for identifying the cultural affinity of names in themselves, some benefit can be gained from combining the judgments of these two modules when neither has been successful in reaching a conclusion within a reasonable level of certainty on its own.

3.1.12.3.2. Additionally, within the CLASS-E system, information about the Country of Birth (COB)

will usually be available. Although this information is not generally sufficient to determine the cultural affinity of a name in itself, it could provide the additional evidence necessary to reach a conclusion when combined with the judgments of the LID and the Digraph Distribution Processor.

- 3.1.12.3.3. The final decision processor exists to take all of this information into account, in an effort to determine the cultural affinity of the input name by combining all available data when the individual data elements themselves are not strong enough indicators.

3.1.12.4. Function

- 3.1.12.4.1. In the event that neither the LID nor the Digraph Distribution Processor is successful in determining a cultural affinity for the input name, the processing continues to the Final Decision Processor. (See sections 3.1.2.4.10, 3.1.8.4.5, and 3.1.9.4.8.)
- 3.1.12.4.2. If the options suggested in sections 3.1.1.4.1.1.1, 3.1.8.4.4, and 3.1.11.4.4 are incorporated into the implementation, the final Decision Processor will have access to all of the information it needs to perform its task encapsulated in the name information object. Otherwise, the Final Decision Processor will take as input LID scores (for each cultural affinity and for Other) and digraph scores (for each cultural affinity) for the input name.
- 3.1.12.4.3. For each culture still under consideration, the final decision processor will determine if the Digraph Distribution score for that culture is within the range specified by the Under_Di_Threshold parameter¹. Note that since there is no Digraph Distribution score calculated for the cultural affinity "Other",

¹ For additional information regarding the range specified by the Under_Di_Threshold parameter, see section 3.2.9.4.2.5.

there is no Under_Di_Threshold parameter associated with Other, and this processing applies only to cultures included in the current name classifier (e.g. Arabic and Hispanic).

3.1.12.4.3.1. In the event that the Digraph Distribution score is in the range specified for the particular culture, processing continues to determine if there is enough additional evidence to identify the input name as belonging to that culture.

3.1.12.4.3.2. In the event that the Digraph Distribution score is not in the range specified for the particular culture, that cultural affinity is removed from further consideration for the input name.

3.1.12.4.4. For each culture still under consideration, the final decision processor will determine if the LID score for that culture is within the range specified by the Under_LID_Threshold parameter¹.

3.1.12.4.4.1. In the event that the LID score is in the range specified for the particular culture, the final decision processor will identify the input name as belonging to that culture.

3.1.12.4.4.2. In the event that the LID score is not in the range specified for the particular culture, processing continues to determine if there is enough additional evidence to identify the input name as belonging to that culture.

3.1.12.4.5. For each culture still under consideration, the Final Decision Processor determines whether

¹ For more information regarding the range specified by the Under_LID_Threshold parameter, see section 3.2.9.4.2.4.

the COB supplied with the input name is in the partition associated with the cultural affinity, as defined in the COB Proximity (COBPROX) Data Store.

3.1.12.4.5.1. In the event that the COB supplied with the input name is in the partition associated with the cultural affinity under consideration, the final decision processor will identify the input name as belonging to that culture.

3.1.12.4.5.2. In the event that the COB supplied with the query is not in the partition associated with the cultural affinity under consideration, that cultural affinity is removed from further consideration for the input name.

3.1.12.4.5.3. In the event that the COB supplied with the input name is Unknown (i.e. "XXX" in Legacy CLASS), the Final Decision Processor will identify the input name as belonging to the cultural affinity under consideration. Note that this is a conscious decision to err on the side of recall in the absence of adequate information (that is, to identify a name as belonging to a culture, perhaps erroneously, in an effort to avoid erroneously not identifying some input names as belonging to that culture). This is related to the other policy decisions to be made by CA, and may change based on those decisions.

3.1.12.4.6. A summarization of the processing performed by the final decision processor is contained in Figure 3-2.

```
IF (Di_Threshold - Digraph_Distribution_Score - Under_Di_Threshold >= 0) AND  
  ((LID_Threshold - LID_Score - Under_LID_Threshold >= 0) OR  
   (COB_of_input_name is in partition OR COB_of_input_name is Unknown))  
THEN  
  Identify Input Name as belonging to the culture in question  
END IF
```

Figure 3-2

3.1.12.5. Subordinates

None.

3.2. Data Decomposition

The data tables which underlie the Linguistically Informed Decision processor are crucial to the success of the algorithm. As discussed in 3.1.2.4.3, the linguistic data to be used are: High Frequency names, TAQ elements, Morphological elements and Ngrams. The entries for each of these linguistic sources will be associated, minimally, with a name field, a cultural group and a score. Also associated with the LID are control parameters. The Data entities accessed by the LID, as well as by other ANC-E Modules are depicted in Figure 3-3. This section describes in detail the data stores used by ANC-E. For examples of the type of information to be included in the data stores, see the detailed example in Appendix A.

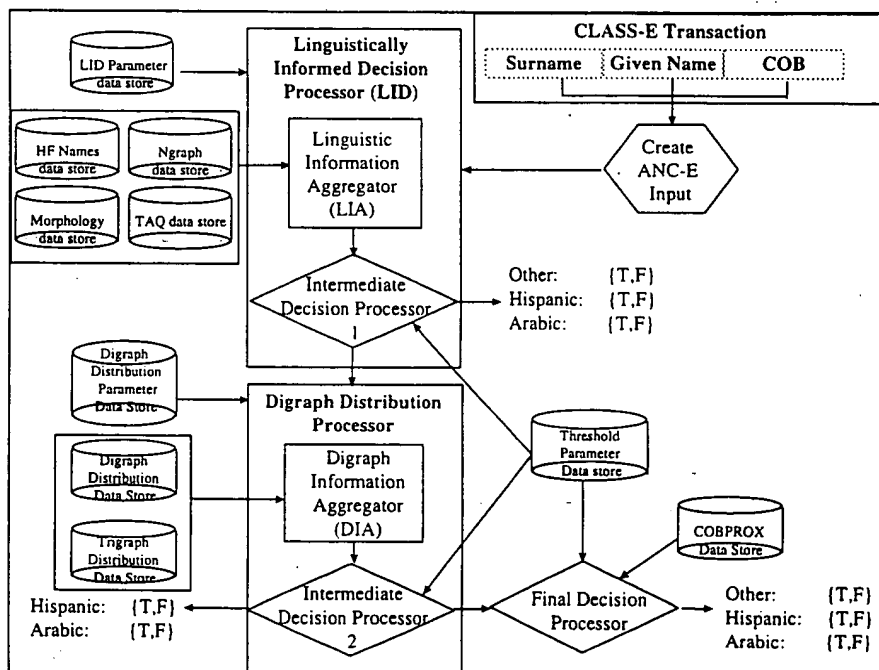


Figure 3-3

3.2.1. LID Parameter Data Store Data Decomposition

3.2.1.1. Identification

This data store is referred to as the LID Parameter Data Store.

3.2.1.2. Type

The LID Parameter Data Store is a data store that is accessed by the LID module.

3.2.1.3. Purpose

3.2.1.3.1. The LID takes factors such as HF names, Ngrams, TAQ particles, and Morphemes into account in determining the cultural affinity of the input name.

3.2.1.3.2. Although each of these factors is valuable, they should not all be given the same relative weight in determining the cultural affinity score of the input name.

3.2.1.3.3. Furthermore, as in all real-world applications, the data in the CLASS-E database is not "clean". That is, data elements are not always found in the expected positions. Therefore, it is common to find surname elements in the given name field, and vice versa. Since it is not always possible to determine whether a particular instance of "out-of-field" data is due to random factors influencing data entry procedures or to a name's being from a culture other than the one hypothesized, data found "out of position" should not be given as great a weight as data found in the canonical position.

3.2.1.3.4. The LID Parameter data store exists in order to allow for different weighting of evidence found by the LID based on the above factors without hard-coding the exact weighting scheme itself in the LID. This will allow for runtime fine-tuning and adjustments to ANC-E without the necessity of recompiling LID module code.

3.2.1.4. Function

3.2.1.4.1. Table 3.2-1 contains a description of the data to be contained in the LID Parameter data store.

DATA NAME	DATA TYPE	DATA WIDTH	POSSIBLE VALUES
AGENT_NAME	character	10	{HFNAME, TAQ, MORPHOLOGY, NGRAM}
NAMEFIELD	character	1	{G,S}
INFIELD_SCORE	integer	2	{1,2,..., 10}
OUT_OF_FIELD_SCORE	integer	2	{1, 2,..., 10}

Table 3.2-1

3.2.1.4.2. The LID uses the information provided in this data store when calculating aggregate cultural affinity scores from the detailed scoring information returned by LIA.

3.2.1.4.2.1. AGENT_NAME indicates to which agent (function) the given INFIELD_SCORE and OUT_OF_FIELD_SCORE weightings apply.

3.2.1.4.2.2. NAMEFIELD indicates whether the INFIELD_SCORE and OUT_OF_FIELD_SCORE weightings apply to the Given Name (G) or to the Surname (S).

3.2.1.4.2.3. IN_FIELD_SCORE is the weighting to be applied to data elements' raw scores returned by the specified agent when found in the specified name field.

3.2.1.4.2.4. OUT_OF_FIELD_SCORE is the weighting to be applied to data elements' raw scores returned by the specified agent when found out of the specified name field.

3.2.1.4.2.4.1. For more information concerning IN_FIELD and OUT_OF_FIELD attributes returned from individual agents via LIA, see 3.1.4.4.3.

3.2.1.4.2.4.2. For an example of scoring of an input name using raw scores returned by agents and the LID Parameters, see Figure 3-4.

3.2.1.5. Subordinates
None.

3.2.2. High Frequency Name Data Store Data Decomposition

3.2.2.1. Identification
This data store is referred to as the HF Name Data Store.

3.2.2.2. Type
The HF Name Data Store is a data store that is accessed by the HF Name Processor.

3.2.2.3. Purpose
The HF Name Data Store encodes the knowledge necessary for the HF Name Processor function of the LID to add information needed for the cultural identification of the input name.

3.2.2.4. Function

3.2.2.4.1. Table 3.2-2 contains a description of the data to be contained in the HF Name data store.

DATA NAME	DATA TYPE	DATA WIDTH	POSSIBLE VALUES
NAME	character	24	*
NAMEFIELD	character	1	{G,S }
SCORE	integer	1	{1,2,3,4,5}
CULTURE	character	1	{H, A, O}

Table 3.2-2

3.2.2.4.2. The HF Name Processor uses the information provided in this data store when gathering detailed HF name cultural affinity information to be returned to LIA. High frequency given names and surnames for each of the three target cultural groups will be listed in the high frequency data store.

3.2.2.4.2.1. NAME indicates the literal string representation of the HF name.

3.2.2.4.2.2. NAMEFIELD indicates whether the score listed for the HF name applies to the Given Name (G) or to the Surname (S).

3.2.2.4.2.3. SCORE reflects the degree to which a name may be considered high frequency within the culture in question, and is the score assigned by the HF Name Processor when the HF name listed is found in the input name. For processing details, see section 3.1.4, High Frequency (HF) Name Processor Module Decomposition.

3.2.2.4.2.4. CULTURE indicates the cultural affinity with which the given NAME-NAMEFIELD-SCORE combination is associated.

3.2.2.4.2.5. A HF name string may appear in the HF Names Data Store multiple times if it is associated with multiple cultural affinities, or if it associated with a different frequency score in the given name and surname. In this instance, the correct score must be assigned for each CULTURE, NAMEFIELD combination associated with the HF name in question.

3.2.2.4.2.5.1. For an example of scoring of an input name using raw scores returned by agents and

the LID Parameters, see
Figure 3-4.

3.2.2.5. Subordinates

None.

3.2.3. TAQ Data Store Data Decomposition

3.2.3.1. Identification

This data store is referred to as the TAQ Data Store.

3.2.3.2. Type

The TAQ Data Store is a data store that is accessed by the
TAQ Processor.

3.2.3.3. Purpose

The TAQ Data Store encodes the knowledge necessary for
the TAQ Processor function of the LID to add information
needed for the cultural identification of the input name.

3.2.3.4. Function

3.2.3.4.1. Table 3.2-3 contains a description of the data to
be contained in the TAQ data store.

DATA NAME	DATA TYPE	DATA WIDTH	POSSIBLE VALUES
TAQ	character	24	*
NAMEFIELD	character	1	{G,S,B}
SCORE	integer	10	{1,2,3,4,5}
CULTURE	integer	3	1..1,000

Table 3.2-3

3.2.3.4.2. The TAQ Processor uses the information
provided in this data store when gathering
detailed TAQ - cultural affinity information to
be returned to LIA. TAQ values for each of the
three target cultural groups will be listed in the
TAQ data store.

3.2.3.4.2.1., TAQ indicates the literal string
representation of the Title, Affix or
Qualifier particle. Note that only free

Morphemes are included in the TAQ data store, so, by definition, all TAQs are implicitly bounded by white space.

3.2.3.4.2.2. NAMEFIELD indicates whether the score listed for the given TAQ particle applies to the Given Name (G), to the Surname (S), or to Both (B).

3.2.3.4.2.2.1. In the event that the NAMEFIELD is listed as "B", the associated TAQ is defined as "in position" whether it is found in the given name or in the surname field in the input name, and is scored accordingly.

3.2.3.4.2.3. SCORE is a score for the given TAQ-NAMEFIELD-CULTURE combination. The TAQ scores will reflect the predictive value of the TAQ particle for the culture with which it is associated. This is the score assigned by the TAQ Processor when the TAQ particle listed is found in the input name. For processing details, see section 3.1.5, Title, Affix, Qualifier (TAQ) Processor Module Decomposition.

3.2.3.4.2.4. CULTURE indicates the cultural affinity with which the given TAQ-NAMEFIELD-SCORE combination is associated.

3.2.3.4.2.5. A TAQ particle may appear in the TAQ Data Store multiple times if it is associated with multiple cultural affinities. In this instance, the correct score must be assigned for each cultural affinity associated with the TAQ value in question.

3.2.3.4.2.5.1. For an example of scoring of an input name using raw

scores returned by agents and the LID Parameters, see Figure 3-4.

3.2.3.5. Subordinates

None.

3.2.4. Morphological Data Store Data Decomposition

3.2.4.1. Identification

This data store is referred to as the Morphological Data Store.

3.2.4.2. Type

The Morphological Data Store is a data store that is accessed by the Morphological Processor.

3.2.4.3. Purpose

The Morphological Data Store encodes the knowledge necessary for the Morphological Processor function of the LID to intelligently process the input name, evaluating evidence based on culturally-specific Morphemes, and adding this to information needed for the cultural identification of the input name.

3.2.4.4. Function

3.2.4.4.1. Table 3.2-4 contains a description of the data to be contained in the Morphology data store.

DATA NAME	DATA TYPE	DATA WIDTH	POSSIBLE VALUES
MORPHEME	character	24	*
NAMEFIELD	character	1	{G, S, B}
MORHTYPE	character	1	{E, P, S, I, A}
SCORE	integer	1	{1, 2, 3, 4, 5}
CULTURE	character	1	{A, H, O}

Table 3.2-4

3.2.4.4.2. The Morphological Processor uses the information provided in this data store when gathering detailed Morpheme - cultural affinity

information to be returned to LIA. Morpheme values for each of the three target cultural groups will be listed in the Morphological Data Store.

3.2.4.4.2.1. MORPHEME indicates the literal string representation of the Morpheme. Note that only bound Morphemes are included in the Morphological data store, so, by definition, all Morphemes are intended to be located as substrings of individual segments of the input name.

3.2.4.4.2.2. NAMEFIELD indicates whether the score listed for the given Morpheme applies to the Given Name (G), to the Surname (S), or to Both (B).

3.2.4.4.2.2.1. In the event that the NAMEFIELD is listed as "B", the associated MORPHEME is defined as "in position" whether it is found in the given name or in the surname field in the input name, and is scored accordingly.

3.2.4.4.2.3. MORPHTYPE indicates the linguistic distribution of the MORPHEME.

3.2.4.4.2.3.1. Prefixes (P) are substrings which begin in the first character position of a name segment.

3.2.4.4.2.3.2. INFIXES (I) are substrings which begin in a character position in the name segment which is not the first, and end in a character position in the name segment that is not the last. They are substrings that are neither at the beginning nor the end of the name segment.

- 3.2.4.4.2.3.3. SUFFIXES (S) are substrings which end in the final character position of a name segment.
- 3.2.4.4.2.3.4. A MORPHEME for which the MORPHTYPE is indicated as EDGE (E) can be found as either a PREFIX or a SUFFIX in a name segment in the input name.
- 3.2.4.4.2.3.5. A MORPHEME for which the MORPHTYPE is indicated as ALL (A) can be found anywhere in a name segment in the input name.
- 3.2.4.4.2.3.6. MORPHEMES that are found in positions other than those indicated by the corresponding MORPHTYPE are not assigned any points for the purpose of identifying the cultural affinity of the input name.
- 3.2.4.4.2.4. SCORE is a score for the given MORPHEME-NAMEFIELD-MORPHTYPE-CULTURE combination. The MORPHEME scores will reflect the predictive value of the Morpheme for the culture with which it is associated. This is the score assigned by the Morphological Processor when the Morpheme listed is found in the input name. For processing details, see section 3.1.6, Morphological Processor Module Decomposition.
- 3.2.4.4.2.5. CULTURE indicates the cultural affinity with which the given MORPHEME-MORPHTYPE-

NAMEFIELD-SCORE combination is associated.

3.2.4.4.2.6. A Morpheme may appear in the Morphological Data Store multiple times if it is associated with multiple cultural affinities, or if it can be associated with multiple values of NAMEFIELD and/or MORPHTYPE for a given cultural affinity. In this instance, the correct score must be assigned for each MORPHEME-MORPHTYPE-NAMEFIELD-CULTURE combination associated with the Morpheme in question.

3.2.4.4.2.6.1. For an example of scoring of an input name using raw scores returned by agents and the LID Parameters, see Figure 3-4.

3.2.4.5. Subordinates

None.

3.2.5. Ngram Data Store Data Decomposition

3.2.5.1. Identification

This data store is referred to as the Ngram Data Store.

3.2.5.2. Type

The Ngram Data Store is a data store that is accessed by the Ngram Processor.

3.2.5.3. Purpose

The Ngram Data Store encodes the knowledge necessary for the Ngram Processor function of the LID to add evidence based on the distribution of culturally salient Ngrams to information needed for the cultural identification of the input name.

3.2.5.4. Function

3.2.5.4.1. Table 3.2-5 contains a description of the data to be contained in the Ngram data store.

DATA NAME	DATA TYPE	DATA WIDTH	POSSIBLE VALUES
NGRAM	character	10	*
NAMEFIELD	character	1	{G, S, B}
NGRAMTYPE	character	1	{E, P, I, S, A}
SCORE	integer	1	{1, 2, 3, 4, 5}
CULTURE	character	1	{A, H, O}

Table 3.2-5

3.2.5.4.2. The Ngram Processor uses the information provided in this data store when gathering detailed cultural affinity information to be returned to LIA. Ngram values for each of the three target cultural groups will be listed in the Ngram Data Store.

3.2.5.4.2.1. NGRAM indicates the literal string representation of the Ngram. Note that all Ngrams are intended to be located as substrings of individual segments of the input name.

3.2.5.4.2.2. NAMEFIELD indicates whether the score listed for the given Ngram applies to the Given Name (G), to the Surname (S), or to Both (B).

3.2.5.4.2.2.1. In the event that the NAMEFIELD is listed as "B", the associated NGRAM is defined as "in position" whether it is found in the given name or in the surname field in the input name, and is scored accordingly.

3.2.5.4.2.3. NGRAMTYPE indicates the linguistic distribution of the NGRAM.

3.2.5.4.2.3.1. PREFIXES (P) are substrings which begin in the first character position of a name segment.

3.2.5.4.2.3.2. INFIXES (I) are substrings which begin in a character position in the name segment which is not the first, and end in a character position in the name segment that is not the last. They are substrings that are neither at the beginning nor the end of the name segment.

3.2.5.4.2.3.3. SUFFIXES (S) are substrings which end in the final character position of a name segment.

3.2.5.4.2.3.4. An NGRAM for which the NGRAMTYPE is indicated as EDGE (E) can be found as either a PREFIX or a SUFFIX in a name segment in the input name.

3.2.5.4.2.3.5. An NGRAM for which the NGRAMTYPE is indicated as ALL (A) can be found anywhere in a name segment in the input name.

3.2.5.4.2.3.6. NGRAMs that are found in positions other than those indicated by the corresponding NGRAMTYPE are not assigned any points for the purpose of identifying the cultural affinity of the input name.

3.2.5.4.2.4. SCORE is a score for the given NGRAM-NAMEFIELD-NGRAMTYPE-CULTURE combination. The NGRAM scores will reflect the predictive value of the Ngram for the culture with which it is

associated. This is the score assigned by the Ngram Processor when the given Ngram is found in the input name. For processing details, see section 3.1.7, Ngram Processor Module Decomposition.

3.2.5.4.2.5. CULTURE indicates the cultural affinity with which the given NGRAM-NGRAMTYPE-NAMEFIELD-SCORE combination is associated.

3.2.5.4.2.6. An Ngram may appear in the Ngram Data Store multiple times if it is associated with multiple cultural affinities, or if it can be associated with multiple values of NAMEFIELD and/or NGRAMTYPE for a given cultural affinity. In this instance, the correct score must be assigned for each NGRAM-NGRAMTYPE-NAMEFIELD-CULTURE combination associated with the Ngram in question.

3.2.5.4.2.6.1. For an example of scoring of an input name using raw scores returned by agents and the LID Parameters, see Figure 3-4.

3.2.5.5. Subordinates

None.

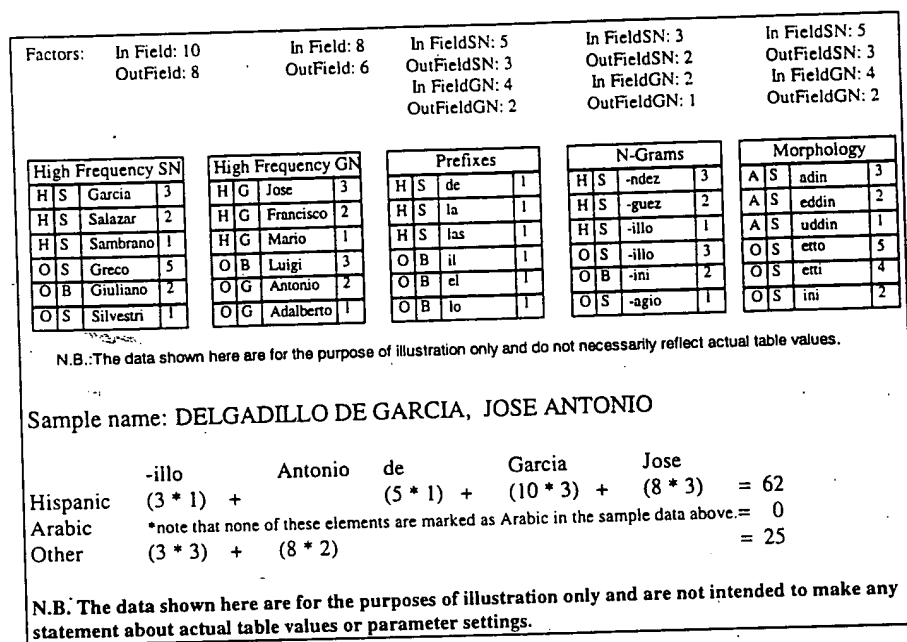


Figure 3-4

3.2.6. Digraph Distribution Data Store Data Decomposition

3.2.6.1. Identification

This data store is referred to as the Digraph Data Store.

3.2.6.2. Type

The Digraph Data Store is a data store that is accessed by the Digraph Distribution Processor.

3.2.6.3. Purpose

The Digraph Data Store encodes the knowledge necessary regarding the statistical distribution of digraphs within a given culture. It is this information that drives the Digraph Distribution Processor.

3.2.6.4. Function

3.2.6.4.1. Table 3.2-6 contains a description of the data to be contained in the Digraph data store.

DATA NAME	DATA TYPE	DATA WIDTH	POSSIBLE VALUES
DI	character	2	*
SCORE	long	3.4	{-50.0000 - +50.0000}
CULTURE	character	1	{A, H}

Table 3.2-6

3.2.6.4.2. The Digraph Processor uses the information provided in this data store when determining the contribution that the distribution of digraphs in the input name will have in determining the cultural affinity of that name. Digraph Distribution statistics will be listed in the Digraph Data Store for each of the specific cultures being identified. That is, in the current implementation, Digraph Distribution statistics will be listed for Arabic and Hispanic; but not for "Other".

3.2.6.4.2.1.1. DI indicates the literal string representation of the digraph. Note that digraphs may include all alphabetical characters as well as the word-boundary character "#".

3.2.6.4.2.2. SCORE reflects the predictive value of the digraph for the culture with which it is associated. This is the score used by the Digraph Distribution Processor when the given digraph is found in the input name. For processing details, see section 3.1.9, Digraph Distribution Processor Module Decomposition.

3.2.6.4.2.3. CULTURE indicates the cultural affinity with which the given DI-SCORE combination is associated.

3.2.6.5. Subordinates
None.

3.2.7. Trigraph Distribution Data Store Data Decomposition

3.2.7.1. Identification

This data store is referred to as the Trigraph Data Store.

3.2.7.2. Type

The Trigraph Data Store is a data store that is accessed by the Digraph Distribution Processor. The Digraph Distribution Processor takes initial and final trigraphs into account in producing a digraph distribution score for the input name.

3.2.7.3. Purpose

The Trigraph Data Store encodes the knowledge necessary regarding the statistical distribution of trigraphs within a given culture. This information is taken into account in the Digraph Distribution Processor, since name boundaries tend to be highly indicative of the cultural affinity of the name.

3.2.7.4. Function

3.2.7.4.1. Table 3.2-7 contains a description of the data to be contained in the Trigraph data store.

DATA NAME	DATA TYPE	DATA WIDTH	POSSIBLE VALUES
TRI	character	3	*
SCORE	long	3.4	{-50.0000 - +50.0000}
CULTURE	character	1	{A, H}

Table 3.2-7

3.2.7.4.2. The Digraph Processor uses the information provided in this data store when determining the contribution that the distribution of initial and final trigraphs in the input name will have in determining the cultural affinity of that name. Trigraph Distribution statistics will be listed in the Trigraph Data Store for each of the specific cultures being identified. That is, in the current implementation, Trigraph Distribution statistics will be listed for Arabic and Hispanic, but not for "Other".

3.2.7.4.2.1. DI indicates the literal string representation of the trigraph. Note that trigraphs may include all alphabetical characters as well as the word-boundary character "#".

3.2.7.4.2.2. SCORE reflects the predictive value of the trigraph for the culture with which it is associated. This is the score used by the Digraph Distribution Processor when the given trigraph is found in the input name. For processing details, see section 3.1.9, Digraph Distribution Processor Module Decomposition.

3.2.7.4.2.3. CULTURE indicates the cultural affinity with which the given DI-SCORE combination is associated.

3.2.7.4.3. Trigraph Distribution statistics for only initial and final trigraphs will be included in the Trigraph Data Store.

3.2.7.5. Subordinates
None.

3.2.8. Digraph Distribution Processor Parameter Data Store Data Decomposition

3.2.8.1. Identification

This data store is referred to as the Digraph Processor Parameter Data Store.

3.2.8.2. Type

The Digraph Processor Parameter Data Store is a data store that is accessed by the Digraph Distribution Processor.

3.2.8.3. Purpose

The Digraph Processor Parameter Data Store contains adjustments that must be made to the digraph distribution scores computed by the Digraph Distribution Processor due to the fact that some cultures are over-represented in the digraph model.

3.2.8.4. Function

3.2.8.4.1. Table 3.2-8 contains a description of the data to be contained in the Trigraph data store.

DATA NAME	DATA TYPE	DATA WIDTH	POSSIBLE VALUES
SKEW	integer	3	{-999 - +999}
CULTURE	character	1	{A, H}

Table 3.2-8

3.2.8.4.2. The Digraph Processor uses the information provided in this data store when determining the final digraph distribution score to assign to the input name. A SKEW will be specified in the Digraph Processor Parameter Data Store for each of the specific cultures being identified. That is, in the current implementation, a SKEW will be listed for Arabic and Hispanic, but not for "Other".

3.2.8.4.2.1. SKEW indicates the value to be added to or subtracted from the raw digraph distribution score by the digraph distribution processor to level data distribution differences.

3.2.8.4.2.2. CULTURE indicates the cultural affinity with which the given SKEW is associated.

3.2.8.5. Subordinates

None.

3.2.9. Threshold Parameter Data Store Data Decomposition

3.2.9.1. Identification

This data store is referred to as the Threshold Parameter Data Store.

3.2.9.2. Type

The Threshold Parameter Data Store is a data store that is accessed by the Intermediate Decision Processor 1 (IDP1),

the Intermediate Decision Processor 2 (IDP2), and the Final Decision Processor.

3.2.9.3. Purpose

The Threshold Parameter Data Store contains information regarding thresholds that must be met in order for the input name to be identified as belonging to a particular target culture.

3.2.9.4. Function

3.2.9.4.1. Table 3.2-9 contains a description of the data to be contained in the Threshold Parameter data store.

DATA NAME	DATA TYPE	DATA WIDTH	POSSIBLE VALUES
CULTURE	character	1	{A, H, O}
LID_THRESHOLD	integer	3	{0 - 999}
DI_THRESHOLD	float	3.4	{-999.9999 - +999.9999}
UNDER_LID_THRESHOLD	integer	3	{0 - 999}
UNDER_DI_THRESHOLD	integer	3	{0 - 999}

Table 3.2-9

3.2.9.4.2. The three "decision processor" modules (IDP1, IDP2, and the Final Decision Processor) use the information provided in this data store when determining whether enough information has been accumulated to identify the input name as belonging to a particular culture. LID_THRESHOLD and UNDER_LID_THRESHOLD data values will be specified in the Threshold Parameter Data Store for each of the cultures being identified, including "Other". DI_THRESHOLD and UNDER_DI_THRESHOLD values will be specified for specific cultures only (i.e. Hispanic and Arabic).

- 3.2.9.4.2.1. CULTURE indicates the cultural affinity with which the given threshold is associated.
- 3.2.9.4.2.2. LID_THRESHOLD is used by IDP1 in determining whether enough information has been accumulated to identify the input name as belonging to a particular culture. For processing information, see section 3.1.8, Intermediate Decision Processor 1 (LID Decision) Module Decomposition.
- 3.2.9.4.2.3. DI_THRESHOLD is used in IDP2 in determining whether enough information has been accumulated to identify the input name as belonging to a particular culture. For processing information, see section 3.1.11, Intermediate Decision Processor 2 (Digraph Decision) Module Decomposition.
- 3.2.9.4.2.4. UNDER_LID_THRESHOLD is used by the Final Decision Processor, and indicates the amount by which a name can fall short of the LID_THRESHOLD and still be considered for membership in a particular culture, provided that other criteria are met. As such, UNDER_LID_THRESHOLD defines a range of values (between the UNDER_LID_THRESHOLD and the LID_THRESHOLD) that, when considered in conjunction with other evidence, can result in the input name's being identified as belonging to the culture in question. For processing information see section 3.1.12, Final Decision Processor Module Decomposition and Figure 3-2.

3.2.9.4.2.5. UNDER_DI_THRESHOLD is used by the Final Decision Processor, and indicates the amount by which a name can fall short of the DI_THRESHOLD and still be considered for membership in a particular culture, provided that other criteria are met. As such, UNDER_DI_THRESHOLD defines a range of values (between the UNDER_DI_THRESHOLD and the DI_THRESHOLD) that, when considered in conjunction with other evidence, can result in the input name's being identified as belonging to the culture in question. For processing information, see section 3.1.12, Final Decision Processor Module Decomposition and Figure 3-2.

3.2.9.5. Subordinates

None.

3.2.10. COB Proximity (COBPROX) Data Store Data Decomposition

3.2.10.1. Identification

This data store is referred to as the COBPROX Data Store.

3.2.10.2. Type

The COBPROX Data Store is a data store that is accessed by the Final Decision Processor.

3.2.10.3. Purpose

The COBPROX Data Store contains information enabling the Final Decision Processor to determine which COBs are to be considered as related when determining the cultural affinity of the input name. For processing information, see section 3.1.12.4.5 and Figure 3-2.

3.2.10.4. Function

3.2.10.4.1.1. ANC-E will use the CLASS-E COBPROX Data Store ("partition table") to fill this function.

3.2.10.5. Subordinates
None.

APPENDIX A: DETAILED EXAMPLE OF ANC-E LID PROCESSING

LIA Data Stores

High Frequency				N-Grams			
H	S	Garcia	3	H	S	S	ndez 3
H	S	Salazar	2	H	S	G	far 1
H	S	Sambrano	1	A	B	P	hous 1
A	B	Mahmoud	4	A	B	S	fiq 2
A	B	Jaffar	2	O	S	E	agio 1
O	S	Silvestri	1	O	S	I	ahmo 5

Morphology				TAO			
A	B	S	addin 2	H	S	de	1
A	B	S	edin 3	H	S	la	1
A	B	S	uddin 3	H	S	las	1
O	S	P	etto 1	A	B	hin	3
O	S	S	etti 2	O	B	el	1
O	S	S	ini 1	O	B	lo	1

Digraph Distribution Processor Data Stores

Digraphs		
A	ez	-1.0422
A	nt	22.8733
A	bd	38.7221
H	bd	1.0572
H	ez	42.5947
H	ri	16.1242

Trigraphs		
A	ez#	-10.0422
A	#nt	-48.1743
A	bd#	48.4551
H	bd#	-32.1742
H	ez#	47.5327
H	#ri	11.1242

Digraph Processor Parameters	
A	44.2331
H	-32.8765

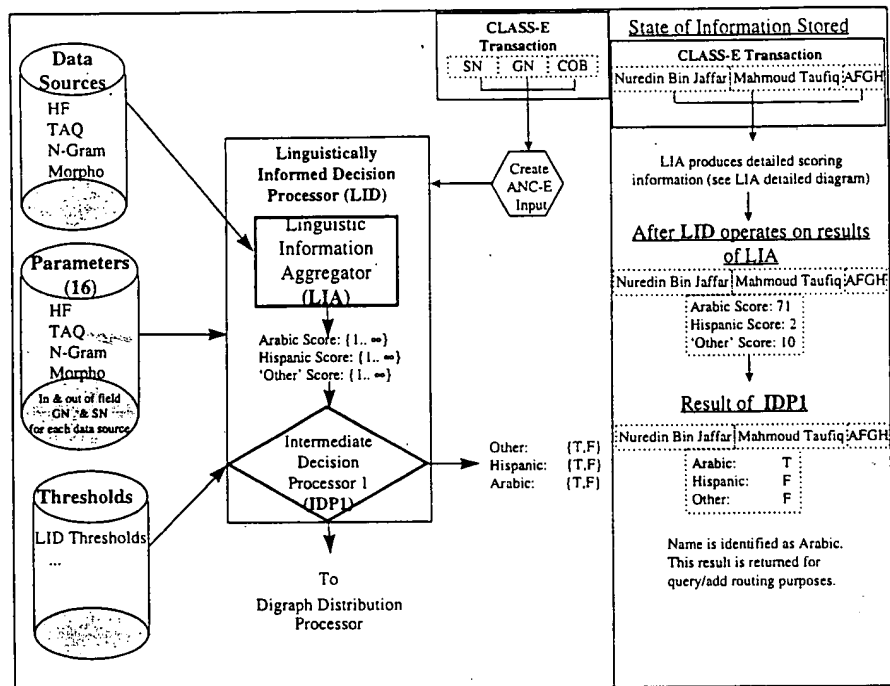
LID Parameters			
HFNAME	S	10	8
HFNAME	G	8	6
TAO	S	5	3
TAO	G	4	2
MORPHOLOGY	S	5	3
MORPHOLOGY	G	4	2
NGRAM	S	4	3
NGRAM	G	3	2

Threshold Parameters			
A	50	1.4532	10 8
H	73	20.5000	5 6
O	38	(null)	100 3

COBPROX	
(CLASS:E.COB Proximity Table to be used for ANC-E.)	

N.B.: The data shown here are for the purpose of illustration only and do not necessarily reflect actual values

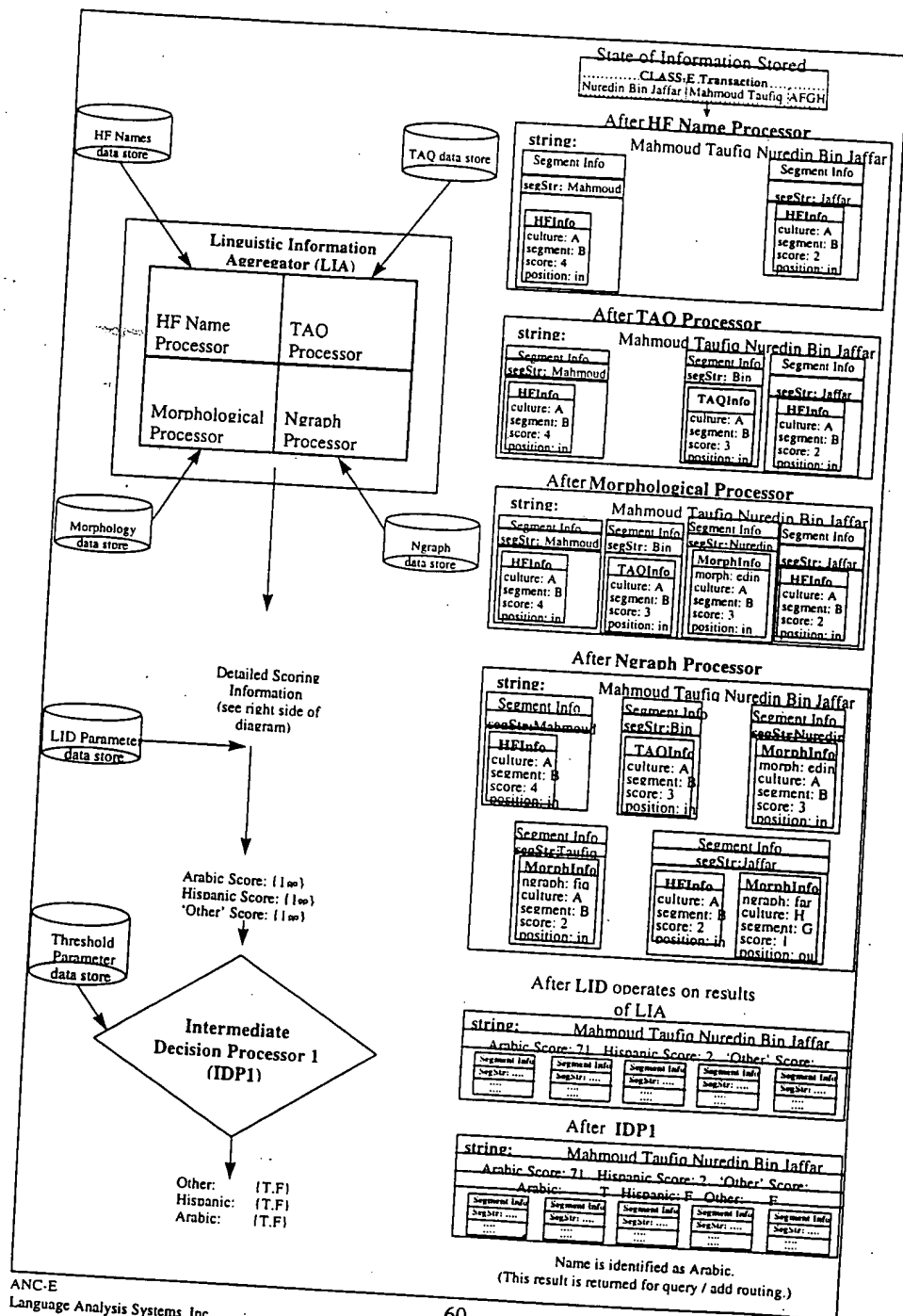
Sample ANC-E Data Stores



Overview of LID Processing

Detailed View of LID Processing
(p. 60)

123456789
101112131415161718192021222324252627282930313233343536373839404142434445464748495051525354555657585960616263646566676869707172737475767778798081828384858687888990919293949596979899100101102103104105106107108109110111112113114115116117118119120121122123124125126127128129130131132133134135136137138139140141142143144145146147148149150151152153154155156157158159160161162163164165166167168169170171172173174175176177178179180181182183184185186187188189190191192193194195196197198199200201202203204205206207208209210211212213214215216217218219220221222223224225226227228229230231232233234235236237238239240241242243244245246247248249250251252253254255256257258259260261262263264265266267268269270271272273274275276277278279280281282283284285286287288289290291292293294295296297298299300301302303304305306307308309310311312313314315316317318319320321322323324325326327328329330331332333334335336337338339340341342343344345346347348349350351352353354355356357358359360361362363364365366367368369370371372373374375376377378379380381382383384385386387388389390391392393394395396397398399400401402403404405406407408409410411412413414415416417418419420421422423424425426427428429430431432433434435436437438439440441442443444445446447448449450451452453454455456457458459460461462463464465466467468469470471472473474475476477478479480481482483484485486487488489490491492493494495496497498499500501502503504505506507508509510511512513514515516517518519520521522523524525526527528529530531532533534535536537538539540541542543544545546547548549550551552553554555556557558559560561562563564565566567568569570571572573574575576577578579580581582583584585586587588589590591592593594595596597598599600601602603604605606607608609610611612613614615616617618619620621622623624625626627628629630631632633634635636637638639640641642643644645646647648649650651652653654655656657658659660661662663664665666667668669670671672673674675676677678679680681682683684685686687688689690691692693694695696697698699700701702703704705706707708709710711712713714715716717718719720721722723724725726727728729730731732733734735736737738739740741742743744745746747748749750751752753754755756757758759760761762763764765766767768769770771772773774775776777778779780781782783784785786787788789790791792793794795796797798799800801802803804805806807808809810811812813814815816817818819820821822823824825826827828829830831832833834835836837838839840841842843844845846847848849850851852853854855856857858859860861862863864865866867868869870871872873874875876877878879880881882883884885886887888889890891892893894895896897898899900901902903904905906907908909910911912913914915916917918919920921922923924925926927928929930931932933934935936937938939940941942943944945946947948949950951952953954955956957958959960961962963964965966967968969970971972973974975976977978979980981982983984985986987988989990991992993994995996997998999100010011002100310041005100610071008100910101011101210131014101510161017101810191020102110221023102410251026102710281029103010311032103310341035103610371038103910401041104210431044104510461047104810491050105110521053105410551056105710581059106010611062106310641065106610671068106910701071107210731074107510761077107810791080108110821083108410851086108710881089109010911092109310941095109610971098109911001101110211031104110511061107110811091110111111121113111411151116111711181119112011211122112311241125112611271128112911301131113211331134113511361137113811391140114111421143114411451146114711481149115011511152115311541155115611571158115911601161116211631164116511661167116811691170117111721173117411751176117711781179118011811182118311841185118611871188118911901191119211931194119511961197119811991200120112021203120412051206120712081209121012111212121312141215121612171218121912201221122212231224122512261227122812291230123112321233123412351236123712381239124012411242124312441245124612471248124912501251125212531254125512561257125812591260126112621263126412651266126712681269127012711272127312741275127612771278127912801281128212831284128512861287128812891290129112921293129412951296129712981299130013011302130313041305130613071308130913101311131213131314131513161317131813191320132113221323132413251326132713281329133013311332133313341335133613371338133913401341134213431344134513461347134813491350135113521353135413551356135713581359136013611362136313641365136613671368136913701371137213731374137513761377137813791380138113821383138413851386138713881389139013911392139313941395139613971398139914001401140214031404140514061407140814091410141114121413141414151416141714181419142014211422142314241425142614271428142914301431143214331434143514361437143814391440144114421443144414451446144714481449145014511452145314541455145614571458145914601461146214631464146514661467146814691470147114721473147414751476147714781479148014811482148314841485148614871488148914901491149214931494149514961497149814991500150115021503150415051506150715081509151015111512151315141515151615171518151915201521152215231524152515261527152815291530153115321533153415351536153715381539154015411542154315441545154615471548154915501551155215531554155515561557155815591560156115621563156415651566156715681569157015711572157315741575157615771578157915801581158215831584158515861587158815891590159115921593159415951596159715981599160016011602160316041605160616071608160916101611161216131614161516161617161816191620162116221623162416251626162716281629163016311632163316341635163616371638163916401641164216431644164516461647164816491650165116521653165416551656165716581659166016611662166316641665166616671668166916701671167216731674167516761677167816791680168116821683168416851686168716881689169016911692169316941695169616971698169917001701170217031704170517061707170817091710171117121713171417151716171717181719172017211722172317241725172617271728172917301731173217331734173517361737173817391740174117421743174417451746174717481749175017511752175317541755175617571758175917601761176217631764176517661767176817691770177117721773177417751776177717781779178017811782178317841785178617871788178917901791179217931794179517961797179817991800180118021803180418051806180718081809181018111812181318141815181618171818181918201821182218231824182518261827182818291830183118321833183418351836183718381839184018411842184318441845184618471848184918501851185218531854185518561857185818591860186118621863186418651866186718681869187018711872187318741875187618771878187918801881188218831884188518861887188818891890189118921893189418951896189718981899190019011902190319041905190619071908190919101911191219131914191519161917191819191920192119221923192419251926192719281929193019311932193319341935193619371938193919401941194219431944194519461947194819491950195119521953195419551956195719581959196019611962196319641965196619671968196919701971197219731974197519761977197819791980198119821983198419851986198719881989199019911992199319941995199619971998199920002001200220032004200520062007200820092010201120122013201420152016201720182019202020212022202320242025202620272028202920302031203220332034203520362037203820392040204120422043204420452046204720482049205020512052205320542055205620572058205920602061206220632064206520662067206820692070207120722073207420752076207720782079208020812082208320842085208620872088208920902091209220932094209520962097209820992100210121022103210421052106210721082109211021112112211321142115211621172118211921202121212221232124212521262127212821292130213121322133213421352136213721382139214021412142214321442145214621472148214921502151215221532154215521562157215821592160216121622163216421652166216721682169217021712172217321742175217621772178217921802181218221832184218521862187218821892190219121922193219421952196219721982199220022012202220322042205220622072208220922102211221222132214221522162217221822192220222122222223222422252226222722282229223022312232223322342235223622372238223922402241224222432244224522462247224822492250225122522253225422552256225722582259226022612262226322642265226622672268226922702271227222732274227522762277227822792280228122822283228422852286228722882289229022912292229322942295229622972298229923002301230223032304230523062307230823092310231123122313231423152316231723182319232023212322232323242325232623272328232923302331233223332334233523362337233823392340234123422343234423452346234723482349235023512352235323542355235623572358235923602361236223632364236523662367236823692370237123722373237423752376237723782379238023812382238323842385238623872388238923902391239223932394239523962397239823992400240124022403240424052406240724082409241024112412241324142415241624172418241924202421242224232424242524262427242824292430243124322433243424352436243724382439244024412442244324442445244624472448244924502451245224532454245524562457245824592460246124622463246424652466246724682469247024712472247324742475247624772478247924802481248224832484248524862487248824892490249124922493249424952496249724982499250025012502250325042505250625072508250925102511251225132514251525162517251825192520252125222523252425252526252725282529253025312532253325342535253625372538253925402541254225432544254525462547254825492550255125522553255425552556255725582559256025612562256325642565256625672568256925702571257225732574257525762577257825792580258125822583258425852586258725882589259025912592259325942595259625972598259926002601260226032604260526062607260826092610261126122613261426152616261726182619262026212622262326242625262626272628262926302631263226332634263526362637263826392640264126422643264426452646264726482649265026512652265326542655265626572658265926602661266226632664266526662667266826692670267126722673267426752676267726782679268026812682268326842685268626872688268926902691269226932694269526962697269826992700270127022703270427052706270727082709271027112712271327142715271627172718271927202721272227232724272527262727272827292730273127322733273427352736273727382739274027412742274327442745274627472748274927502751275227532754275527562757275827592760276127622763276427652766276727682769277027712772277327742775277627772778277927802781278227832784278527862787278827892790279127922793279427952796279727982799280028012802280328042805280628072808280928102811281228132814281528162817281828192820282128222823282428252826282728282829283028312832283328342835283628372838283928402841284228432844284528462847284828492850285128522853285428552856285728582859286028612862286328642865286628672868286928702871287228732874287528762877287828792880288128822883288428852886288728882889289028912892289328942895289628972898289929002901290229032904290529062907290829092910291129122913291429152916291729182919292029212922292329242925292629272928292929302931293229332934293529362937293829392940294129422943294429452946294729482949295029512952295329542955295629572958295929602961296229632964296529662967296829692970297129722973297429752976297729782979298029812982298329842985298629872988298929902991299229932994299529962997299829993000300130023003300430053006300730083009301030113012301330143015301630173018301930203021302230233024302530263027302830293030303130323033303430353036303730383039304030413042304330443045304630473048304930503051305230533054305530563057305830593060306130623063306430653066306730683069307030713072307330743075307630773078307930803081308230833084308530863087308830893090309130923093309430953096309730983099310031013102310331043105310631073108310931103111311231133114311531163117311831193120312131223123312431253126312731283129313031313132313331343135313631373138313931403141314231433144314531463147314831493150315131523153315431553156315731583159316031613162316331643165316631673168316931703171317231733174317531763177317831793180318131823183318431853186318731883189319031913192319331943195319631973198319932003201320232033204320532063207320832093210321132123213321432153216321732183219322032213222322332234322532263227322832293230323132323233323432353236323732383239324032413242324332443245324632473248324932503251325232533254325532563257325832593260326132623263326432653266326732683269327032713272327332743275327632773278327932803281328232833284328532863287328832893290329132923293329432953296329732983299330033013302330333043305330633073308330933103311331233133314331533163317331833193320332133223323332433253326332733283329333033313332333333343335333633373338333933403341334233433344334533463347334833493350335



**SOFTWARE DESIGN DESCRIPTION
FOR THE ARABIC NAME SEARCH
ALGORITHM
FOR CLASS - E
(ANA - E)**

TABLE OF CONTENTS

1. INTRODUCTION	1
2. MODULE DECOMPOSITION	5
3. DATA DECOMPOSITION	37

SOFTWARE DESIGN DESCRIPTION FOR THE ARABIC NAME SEARCH ALGORITHM FOR CLASS - E (ANA - E)

1. INTRODUCTION

1.1. Purpose

The variation that can occur in the transcription of Arabic names into roman representation poses formidable problems for retrieval systems with very large databases that depend solely on standard string-comparison techniques. For example, the following names are transcription variants of the same name: **SALEHUDDINE, IMHEMED and SAALAH EL DEEN, MUHAMMED**. The significant differences in their spellings and in the distribution of white space would virtually preclude any possibility of identifying these names as similar enough to be candidates for retrieval if the usual techniques were applied. The task, then, is to capture the relatedness of these names and to incorporate the insights into their commonality into the retrieval system.

Arabic names are made up of a Given Name (GN) (usually one, although compound names may occur: **SAMIR; MOHAMAD ALI**) and a string of familial (paternal) relations following the GN (**ABD EL KADEER SAMIR ABD EL LATIF**). The string following the GN is generally made up of GNs which are taken from the father, grandfather and other relations. Only in rare cases can any of these segments be identified as a SN, i.e., a name used by every member of the family to signal family membership. The full string following the GN provides crucial information about the individual that is lost if it is sometimes in the GN field and sometimes in the SN field. So, positioning names that occur after the first GN in the SN field provides the opportunity for better matches.

1.2. Scope

In 1996, LAS proposed an initial solution to the problem, the salient feature of which was to level spelling differences and thereby generate one representation for the myriad spellings of a single name. This process is known as *regularization*, a technique that was implemented in the Legacy CLASS system as Legacy ANA. Legacy ANA is a preprocessing module that feeds into the Legacy CLASS search system. The general characteristics of Legacy ANA are:

- 1) Both query and add procedures are identical for the Legacy ANA system.

- 2) A rudimentary Arabic name identifier (ANI) determines if an input name qualifies for handling by the Legacy ANA algorithm. All names that qualify for Legacy ANA handling are also sent to the generic processing module provided by Legacy CLASS and to the DOB processor, when appropriate.
- 3) A set of regularization rules is applied to the Arabic input name, leveling the spelling differences of the name segments to the most common representation of the input name. **IMHEMED** and **MUHAMMED** will both be regularized to the form **MUHAMAD**, for example. Some title/affix/qualifier information may also be removed from the name to focus on the name stem. The regularization rules are rewrite rules that use notation developed solely for this processor. The rule engine necessary for implementation of the regularization rules was also developed specifically to handle the Legacy ANA regularization rules. The output of the regularization component is a regularized form of the input name.
- 4) The output of the regularization component (the regularized form of the name) serves as the input to the generic CLASS search system. CLASS produces standard compressed-name keys, but on the regularized form that is the output of Legacy ANA. CLASS accesses the database records through the keys on the regularized form. (The keys are generated for both queries and adds and are stored with the record when a record is added to the database.)
- 5) A digraph match is then performed on the regularized record and query forms to determine name similarity. The match criteria are those of CLASS.

The ANA-E system is an enhancement of the Arabic name search system that was developed by LAS for the Legacy CLASS system. The principle of name regularization remains the same in ANA-E, although the design and approach of ANA-E are different in a number of important ways.

- 1) An independent Name Classifier (ANC-E) has been developed. (The ANC-E design description is provided as Attachment A in LAS Linguistic Memo CT970044, May 30, 1997) ANC-E will direct input names to the Arabic and/or Hispanic processors. Its functionality is far more sophisticated than the Arabic name typer (ANI). All records will also be directed to the CN pipe of CLASS-E and to the DOB pipe, if appropriate.
- 2) A significant amount of preprocessing of the name takes place in ANA-E that recognizes the unique character of Arabic names, focusing on the leftmost GN as the most stable element in the name and rejoining all other GN segments with their SN partners.
- 3) The regularization rules and rule engine have changed. The rules are represented in standard regular expressions and the format has changed. The rule engine uses different match techniques, is much simpler in its implementation and therefore can be easily applied to other rule sets.
- 4) The output of the regularization rules is a computationally viable form, one that may not be the most common representation of a name (as was a requirement of Legacy ANA).

- 5) The new ANA-E rule language has already allowed a dramatic increase in the amount of regularization that takes place. For those records that qualify as Arabic, a decrease of 13% in the number of different regularized forms from Legacy ANA (as of March 1997) to the proposed ANA-E regularization rules demonstrates the greater flexibility of the ANA-E language. The system must therefore accommodate fewer distinct name-segments, reducing processing time and increasing the scope of retrieval.
- 6) Arabic-specific keys have been generated to account for the nature of Arabic names and at the same time permit some unpredictable variation. The regularization rules account for much of the predictable variation in names but are only incidentally able to accommodate unpredictable variation (e.g., error).
- 7) Retrieval is based on keys that represent a class of pre-determined variants of a name segment and are formed from the GN1 and SN segments.
- 8) Gender has been added as a search criterion to reduce the occurrence of crossed-gender retrievals.
- 9) The filtering techniques used in ANA-E demonstrate much greater granularity and sensitivity to Arabic-specific name characteristics.

The CLASS-E system will support several concurrent record search processes. The Multi-Pipe Architecture (MPA) already in place supports the generic-CLASS search process and a distinct Date-of-Birth process. (Because the Legacy ANA system serves as a preprocessing module that feeds into the generic-CLASS processing pipe, Legacy ANA is not characterized as a separate search pipe.)

In CLASS-E the Multi-Pipe Architecture will be extended to include a distinct Arabic processing pipe, a distinct Hispanic processing pipe as well as perhaps others in the future. An input name may be submitted to more than one processing pipe. It is a business decision of CA/EX/CSD to determine to which and how many pipes to pass a given input name. It is suggested that names classified as Arabic by the Advanced Name Classifier for CLASS-E (ANC-E) be submitted to multiple processors: the generic CLASS-E processing pipe, the DOB processing pipe and the ANA-E processing pipe.

1.3. Definitions and Acronyms

ACOB	Arabic COB Category Data Store
ADE	Arabic Data Evaluator
AFS	Arabic Filter and Sorter
AG	Applicant Gender (user supplied)
AGI	Arabic Gender Identifier
AKG	Arabic Key Generator
ANA - E	Arabic Name Search Algorithm for CLASS - E
ANC - E	Advanced Name Classifier for CLASS - E
ANI	Arabic Name Identifier (Legacy ANA)
ANR	Arabic Name Regularizer
ANT	Arabic Name Type Data Store
APP	Arabic Pre-Processor
ARE	Arabic Rule Engine

ARR	Arabic Regularization Rules Data Store
ASE	Arabic Search Engine
ASP	Arabic Segment Positioner
ATD	Arabic TAQ Data Store
ATP	Arabic TAQ Processor
COBPROX	COB Proximity Data Store
DELETE	Segment will be removed from any further consideration in the name matching process; it will contribute marginally to the filtering process. The segment will <i>not</i> be removed from the record.
DISREGARD	Segment will be removed from consideration in the name retrieval process but will contribute to the filtering and sorting processes.
DI_VAL	Digraph Value
F	Female Gender
FNU	First Name Unknown
FP	Filter Parameter Data Store
GN	Given Name
GN1	Leftmost GN1 segment
GNDR	Gender
GNTHR	Given Name Threshold (Filter)
GN_VAL	Final Given Name Value
Given Name Field	All name segments to the right of the comma
HF	High Frequency
HFI	Arabic High Frequency Name Identifier
HK	High Frequency Key
HS	High Frequency Search Key
INITGN	Given Name Initial Value
INITSN	Surname Initial Value
K-Key	Special Key formed to handle name segments with "k"
Legacy ANA	Arabic Name Algorithm for Legacy CLASS
LF	Low Frequency
LTF	Linguistic Trace Facility
M	Male Gender
OPSN	Out-of-Position Surname Segment
OPVAL	Out-of-Position Value (Filter)
PK	Primary Key
RCL	Refusal Code Level Data Store
Record Gender	Gender determined for a record based on two gender validators, input gender and HF name gender; all gender indicators must agree.
Regularization	Rule-based process that levels the differences among the roman spellings of a single Arabic name
RG	Record Gender
RLYOB	Refusal Code Level/Year-of-Birth Range Data Store
Segment	Any single name piece, surrounded by white space
SI	Single-Part Key
SK	Search Key
SNTHR	Surname Threshold (Filter)
SN_VAL	Final Surname Value
SP	Special Key
SS	Standard Search Key
Surname Field	All name segments to the left of the comma
TF	TAQ Filter Data Store

TAQ	Title/Affix/Qualifier
TAQAGN	Value for Missing TAQ in the Given Name
TAQASN	Value for Missing TAQ in the Surname
TAQXGN	Value for TAQ DELETE in Given Name
TAQXSN	Value for TAQ DELETE in Surname
U	Unknown (Ambiguous) Gender
WK	Wild-Card Key
YR	Year-of-Birth Range Data Store

2. MODULE DECOMPOSITION

2.1. The Arabic Name Search Algorithm for CLASS-E (ANA-E) will consist of three primary components (see pages 6-9 for graphic representations of these components):

- the Arabic Pre-Processor (APP),
- the Arabic Search Engine (ASE), and
- the Arabic Filter and Sorter (AFS).

2.2. ARABIC PRE-PROCESSOR MODULE DECOMPOSITION

2.2.1. Identification

This module is known as the Arabic Pre-Processor (APP).

2.2.2. Type

The APP is the first programming module in the larger ANA-E algorithm and consists of subordinate functions that manipulate the name segments in various ways to prepare the name for creation of search keys by Arabic Search Engine.

2.2.3. Purpose

Because of the significant variation that can occur in names that have been romanized from the original Arabic script, Arabic names will benefit from attempts to level the spelling differences. In addition, the standard format of an Arabic name is Given Name followed by a string of segments that indicate familial relations. In many countries, none of these segments functions as what is standardly referred to as a *surname*. What is determined to be a Surname for purposes of a CLASS retrieval (i.e., what is placed in the Surname field) is therefore quite arbitrary. Arabic names will consequently benefit from movement of name segments that would contribute to a potential match.

2.2.4. Function

2.2.4.1. The Arabic Pre-Processor (APP) will accept as input any name that has been identified as Arabic by the Advanced Name Classifier for CLASS-E (ANC-E) and will prepare a name for the Arabic Search Engine by applying Arabic regularization rules to the name segments and reorganizing the name according to Arabic naming principles.

2.2.4.2. The APP can alternatively create a name object that "knows" characteristics about itself and collects information as it proceeds through the processing functions.

2.2.5. Subordinates

- Arabic Name Regularizer (ANR)
- Arabic TAQ Processor (ATP)
- Arabic Data Evaluator (ADE)
- Arabic Segment Positioner (ASP)
- Arabic Gender Identifier (AGI)

2.3. ARABIC NAME REGULARIZER MODULE DECOMPOSITION

2.3.1. Identification

This module will be known as the Arabic Name Regularizer (ANR) and will consist of one subordinate processor, the Arabic Rule Engine, which will access and apply the rules in one data store, the Arabic Regularization Rules (ARR) Data Store.

2.3.2. Type

The ANR is a program that

- will operate on a full surname string and a full given name string of an add or query record,
- will generate a regularized form for each name segment or string of name segments to which the regularization rules have applied, and
- will submit the regularized form to other functions in the APP to continue to prepare the name for submission to the Arabic Search Engine.

2.3.3. Purpose

The transcription of Arabic names from their native format (Arabic script) to the roman alphabet is highly variable; few, if any, transcription standards exist. Such rampant variation poses significant problems for string matching and retrieval systems; there are often too many characters that differ to effect a retrieval in character-based retrieval systems. For example, **MUHAMMAD** and **IMHEMED** are roman spellings of the same name; that is, they are represented by the same string of characters in the Arabic script. Leveling the differences in roman spelling, wherever possible, would improve record retrieval dramatically.

2.3.4. Function

The ANR applies a set of regularization rules (ARR) to the surname and to the given name through the Arabic Rule Engine and produces a regularized form for any name segment or string of segments to which the rules can apply.

2.3.5. Subordinates

The ANR consists of one subordinate function, the Arabic Rule Engine, which accesses the Arabic Regularization Rule Data Store.

2.4. ARABIC RULE ENGINE MODULE DECOMPOSITION

2.4.1. Identification

This function is known as the Arabic Rule Engine (ARE).

2.4.2. Type

The ARE is a program that attempts to apply transformation rules to an input string of characters and to effect a change in that string.

2.4.3. Purpose

- 2.4.3.1. The development of rules that are implemented in standard and readily accessible regular expressions allows for use of a less idiosyncratic rule engine than the one developed for Legacy ANA. (See Arabic Regularization Rule Data Store, Section 3.3).
- 2.4.3.2. The Arabic Regularization Rules (ARR) require an implementation module to effect the changes specified in the rules. The ARE plays that role.
- 2.4.3.3. The ARE replaces the rule engine developed for Legacy ANA. It is simpler and more generic and can be used for other rule implementations.
- 2.4.3.4. The ARR are more easily altered and reviewed.

2.4.4. Function

- 2.4.4.1. The ARE accepts a full surname (SN) or full given name (GN) string as input.
- 2.4.4.2. The ARE will add a white space to the beginning of the SN or GN string that it accepts to serve as boundary markers.
- 2.4.4.3. The ARE scans the input string from left to right and attempts to match the Match Context of a rule.
- 2.4.4.4. If the ARE is able to identify a Match Context, it checks to see if the Pre- and Post-Contexts specified in the rule are present.
 - 2.4.4.4.1. If the Pre- and Post-Contexts specified in the rule match, then the ARE applies the rule and makes the specified change in the Match Context, producing the Output.
 - 2.4.4.4.2. The ARE then returns to the top of the rule set and attempts to identify a Match Context beginning with the character immediately following the previous Match Context.
- 2.4.4.5. If no match is found, the ARE moves to the Match Context of the next rule.
- 2.4.4.6. If no rule has fired, the default rule applies: the character output is the character itself. E.g., $S \rightarrow S$
- 2.4.4.7. **Arabic Regularization Rules (ARR)**
 - 2.4.4.7.1. The ARRs are written as regular expressions and use, for the most part, regular expression notation. See Section 3.3 for ARR details.
 - 2.4.4.7.2. The ARRs use defined metasympols.
 - 2.4.4.7.3. The ARE must be able to recognize all regular expression notation and metasympols of the ARR and implement them.

2.4.4.7.4. ARRs have the format:

Figure 1: Format: Arabic Regularization Rule

PRE-CONTEXT	MATCH CONTEXT	POST-CONTEXT	→	OUTPUT
-------------	---------------	--------------	---	--------

2.4.4.7.5. Rule Ordering

2.4.4.7.5.1. Rule ordering is important because the **first** rule for which the ARE finds a Match Context (and the Pre- and Post-Contexts match as well) will apply. Once the rule has applied to the Match Context, no other rules will apply to it: *No* following rule will then fire on that Match Context.

2.4.4.7.5.1.1. The rules must have internal ordering based on the Match Context only.

2.4.4.7.5.1.2. Rules may intrude in the ordering of the Match Context if they are applicable to another phenomenon.

2.4.4.7.5.1.2.1. For example, an MI → NE rule will need to precede an M → N rule or the MI → NE will never apply.

2.4.4.7.5.1.2.2. A rule that applies to a Match Context where A → AW could intervene between the "M" rules and have no effect on the order of application of the "M" rules.

2.4.4.7.5.1.2.3. In general, rules with longer character strings in the Match Context need to precede rules with shorter character strings.

2.4.4.7.5.1.2.4. Care must be taken when rules have symbols for optional characters, for example. The ordering of an M?L rule (a rule that can apply to ML or L) must be carefully placed with respect to other rules that apply to M and L.

2.4.4.7.5.2. The Output of one rule does *not* form the input to another rule.

- 2.4.4.7.5.2.1. Only one rule applies to a character or character string that matches a Match Context.
- 2.4.4.7.5.2.2. The first rule that matches all three contexts in the Match, Pre- and Post- Context order is applied.
- 2.4.4.7.5.2.3. The Output of a rule cannot then be changed.
- 2.4.4.7.5.2.4. Rules must be written so that they stand alone: rules are not interdependent.
- 2.4.4.7.5.3. If the ARE is able to match the Match Context, the Pre- and Post-Contexts are examined.
 - 2.4.4.7.5.3.1. If the Pre- and Post-Contexts both match, the ARE effects the change in the Match Context indicated in the OUTPUT.
 - 2.4.4.7.5.3.1.1. The next available context in the input string to be considered for a match immediately follows the previous *Match Context*.
 - 2.4.4.7.5.3.1.2. For example, if HEIMER is the input string and a rule applies to HEIM to make it GIM, the next available context for consideration is the E of ER (following HEIM). If an E rule is to apply, it can only apply to the *second* E, not that of the previous Match Context (HEIM).
 - 2.4.4.7.5.3.1.3. The Output of a previous rule cannot be the Pre- or Post-Context of a following rule.
 - 2.4.4.7.5.3.2. The rule is applied only if the ARE is successful in matching the Match Context and the Pre- and Post-Contexts.
- 2.4.4.7.6. There is no backtracking in the ARE.
- 2.4.4.7.7. The output of successful application of rule(s) by the ARE is a regularized Arabic form. The output of the ARE can be in any string form (e.g., binary, regular expression, characters).

2.4.4.8. Subordinates

None.

2.5. ARABIC TAQ PROCESSOR MODULE DECOMPOSITION

2.5.1. Identification

This function is known as the Arabic TAQ Processor (ATP).

2.5.2. Type

The ATP is a function that identifies titles (T), affixes (A) and qualifiers (Q), as specified in the Arabic TAQ Data Store, and implements the disposition indicated in that table.

2.5.3. Purpose

Arabic names frequently contain peripheral name elements, such as ABDEL, ABU, AL. Matching on these segments is not generally useful; the name segments with information value are the name stems, RAHMAN, SAYED, HANAWI. Removal of or disregard for the peripheral name elements allows more emphasis to be placed on the name stems.

2.5.4. Function

2.5.4.1. The ATP will access the Arabic TAQ Data Store (ATD) to identify titles (e.g., USTAAZ), affixes (e.g., EL DIN) and qualifiers (Q) that occur in the regularized name.

2.5.4.2. The ATP will tag as a T, P, I, S, or Q any such segments found in the name, as specified in the ATD.

2.5.4.2.1. The ATP will scan the full SN or GN field for any TAQ segments.

2.5.4.2.2. If the ATP identifies a segment, it will tag the segment with the ID_NO and disposition, as indicated in the ATD.

2.5.4.2.3. If the following segment is also a TAQ segment, it will tag the segment with the ID_NO and disposition, as indicated in the ATD.

2.5.4.2.4. This will continue until all *consecutive* TAQ segments have been tagged.

2.5.4.2.5. When the ATP encounters a following segment that is not a TAQ segment, it will treat that segment as a stem.

2.5.4.2.5.1. Each TAQ segment identified up to that point will be given the TAQ_TYPE P (prefix) and each will be associated and stored with the following stem.

2.5.4.2.6. The ATP will move to the next segment following the stem and will repeat the TAQ identification process.

2.5.4.2.6.1. The ATP will tag all TAQ segments with the ID_NO and disposition.

2.5.4.2.6.2. When the ATP encounters a stem, it will tag each TAQ segment (not yet associated with a stem) with the TAQ_TYPE P and will associate and store each TAQ segment with the following stem.

2.5.4.2.7. If the ATP encounters a TAQ segment (or segments) that has no following stem, it will access the ATD to determine if the TAQ type is a Suffix (S).

2.5.4.2.7.1. If the TAQ has a TAQ_TYPE S, the TAQ will be associated and stored with the *preceding* stem.

2.5.4.2.7.2. The preceding stem may already have prefixal TAQs.

2.5.4.2.7.3. If the TAQ type is not equal to S, the TAQ will be tagged a Stranded Affix.

2.5.4.3. The ATP will process any TAQ segments identified according to the treatment indicated in the ATD. (See Section 3.4.)

2.5.4.3.1. Treatment options include DELETE and DISREGARD.

2.5.4.3.2. **DELETE** means that the segment is completely disregarded in the remainder of the name search process and contributes marginal information to the filtering process. (N.B. The segment is not deleted from the record.)

2.5.4.3.3. **DISREGARD** means that the segment is disregarded in the remainder of the name search process but contributes to the evaluation of the name in the filtering processes.

2.5.4.4. TAQ Tag

2.5.4.4.1. The TAQ tag will reference the ID_NO of the TAQ.

2.5.4.4.2. The TAQ tag will reference the indicated treatment of the TAQ segment.

2.5.4.4.3. The TAQ tag will be associated with a name stem, unless marked as a Stranded Affix.

2.5.4.4.4. Surnames containing the prefix AL (e.g., AL IDRISI) will be specially marked.

2.5.4.5. The TAQ tag will assist in the sorting of records (see Section 2.12, the Arabic Filter and Sorter (AFS)).

2.5.5. Subordinates

None.

2.6. ARABIC DATA EVALUATOR MODULE DECOMPOSITION

2.6.1. Identification

This function is known as the Arabic Data Evaluator (ADE).

2.6.2. Type

The ADE is a function that "corrects" data entry errors by generating one or more alias records.

2.6.3. Purpose

Arabic names are conventionally a Given Name followed by a string of (usually paternal) relationships, elements of which are routinely deleted. Some data entry operators have apparently attempted to capture the fact that the Arabic name is closer to a single name string by entering XXX into the Given Name field, cf., presumably the XXX permitted in the COB or DOB fields.

Because XXX is not a conventional representation of any Given Name information, it interferes with the name search and will be altered.

2.6.4. Function

The ADE will determine if the leftmost Given Name segment (only) is XXX. If so, it will change that string to FNU and generate an alias add record or query.

2.6.5. Subordinates.

None.

2.7. ARABIC SEGMENT POSITIONER MODULE DECOMPOSITION

2.7.1. Identification

This function is known as the Arabic Segment Positioner (ASP).

2.7.2. Type

The ASP is a processing module that operates on the preprocessed, regularized name and moves name segments from the Given Name field into the Surname field. Alias records will be produced to reflect format changes.

2.7.3. Purpose

Arabic names are made up of a Given Name (GN) (usually one, although compound names may occur: SAMIR; MUHAMAD ALI) and a string of familial (paternal) relations following the GN (ABD EL KADEER SAMIR ABD EL LATIF). This string is generally made up of GNs which are taken from the father, grandfather and other relations. In most cases, none of these segments be identified as a Surname, i.e., a name used by every member of the family to signal family membership. The full string following the GN provides crucial information about the individual that is lost if it is sometimes in the GN field and sometimes in the SN field. So, positioning names that occur after the first GN in the SN field provides the opportunity for better matches.

2.7.4. Function

2.7.4.1. The ASP will move all segments to the right of the leftmost GN (GN1) (in the preprocessed; regularized name) to the leftmost SN position, preserving the order of the moved segments.

Figure 2: Movement of Segments into SN Field

FARUK, MUHAMAD SAMIR ABDULA	→	SAMIR ABDULA FARUK, MUHAMAD
-----------------------------	---	-----------------------------

2.7.4.2. The leftmost Given Name (GN1) segment will not be moved into the Surname field *except*

- if there is one and only one GN segment and
- if there is one and only one SN segment which has been tagged as having the prefix AL,
- then the ASP will generate an alias record with the SN and GN inverted.

Figure 3: Inversion of SN and GN with AL in the SN Field

SURNAME	GN1	
(AL) IDRISI	YUSEF	→
YUSEF	(AL) IDRISI	

2.7.4.3. The GN1 may be a name segment, an initial, or FNU. (See Section 2.9, the Arabic Search Engine (ASE) for additional information.)

2.7.5. Subordinates

None.

2.8. ARABIC GENDER IDENTIFIER MODULE DECOMPOSITION

2.8.1. Identification

This function is known as the Arabic Gender Identifier (AGI).

2.8.2. Type

2.8.2.1. The AGI is a function that will apply after the ANR has produced a regularized representation of the input name and the Arabic Segment Positioner (ASP) has moved all GN segments other than the GN1 into the SN field.

2.8.2.2. For the AGI to derive record gender, the data input operator will need to supply gender for each record added to the database and for each query during the data entry process.

2.8.3. Purpose

2.8.3.1. Crossed-gender records are of little value to the system user.

2.8.3.2. Arabic gender is reliably predictable from the GN1.

2.8.3.3. Records that have crossed gender will receive lowered match values during the filtering and sorting process.

2.8.3.4. Record gender requires gender validation from two sources: gender received during the data entry process and predictable gender associated with Arabic names.

2.8.3.5. Record gender reduces the chance of associating gender with a name that may be misspelled.

2.8.4. Function

2.8.4.1. The AGI will derive the record gender for all record adds and queries.

2.8.4.1.1. For each query and add name, the AGI will derive record gender from user-supplied gender input and from the gender that has been assigned to the GN1.

2.8.4.1.2. A *minimum* of two gender indicators is required for a gender assignment of M or F.

2.8.4.2. For record adds, gender received as input from the data entry process will be stored with the record.

2.8.4.3. For record queries, the user will input the gender of the applicant at query time.

2.8.4.4. For both adds and queries, the AGI will access the Arabic Name Type Data Store (ANT) and will assign the gender value to the GN1 segment, as indicated in the ANT (GENDER). (See Section 3.5.)

2.8.4.4.1. If the name is present in the ANT, the gender associated with the name segment will be compared to the data entry gender.

2.8.4.4.1.1. If the gender indicators match, the matching value will become the record gender.

2.8.4.4.1.2. If the gender indicators do not match, the record gender will be Unknown (U).

2.8.4.4.2. If the name is not present in the ANT, the record gender will be marked as Unknown (U).

2.8.5. Subordinates

None.

2.9. ARABIC SEARCH ENGINE MODULE DECOMPOSITION

2.9.1. Identification

This module is known as the Arabic Search Engine (ASE).

2.9.2. Type

The ASE is a processing module that accepts the output of the Arabic Preprocessor (APP), generates retrieval keys through the Arabic Key Generator, retrieves candidate records from the database based on the keys and submits those candidate records to the Arabic Filter and Sorter (AFS) Module.

2.9.3. Purpose

The regularized, repositioned names generated by the APP will be, in general, a representation of the canonical form of the Arabic name. The search process will benefit from focus on the canonical form of the Arabic name.

2.9.4. Function

The ASE will retrieve records from the database whose stored keys match the keys generated for the query record.

2.9.5. Subordinates

The ASE has one subordinate module:

- Arabic Key Generator

2.10. ARABIC KEY GENERATOR MODULE DECOMPOSITION

2.10.1. Identification

This function is known as the Arabic Key Generator (AKG).

2.10.2. Type

The AKG is a function that will form keys from the GN1 and each SN segment of the preprocessed, regularized name for both record adds and queries.

2.10.3. Purpose

In order to reduce the number of records that must be compared by the Arabic Filter and Sorter Module, it is desirable to subset the Arabic database. (About 500,000 records have qualified as Arabic through the ANI name typing process and it is assumed that this number will continue to represent the approximate size of an Arabic database.) One mechanism for achieving a

subset is to generate keys for the input name. The Arabic keys are motivated by the nature of the Arabic name and are centered around the most stable name segment in the Arabic name, the GN1.

2.10.3.1. The Arabic keys replace the compressed-name keys produced for Legacy ANA, which have severe limitations for retrieving both predictable and unpredictable variants of the regularized Arabic names.

2.10.3.2. For record adds, all keys will be stored with the source record.

2.10.3.3. Keys will be generated for each SN segment (moved or resident) and for each GN1.

2.10.3.3.1. For record adds, more keys will be generated for HF name segments than for LF segments.

2.10.3.3.2. Search keys will be a combination of SN keys and GN1 keys.

2.10.4. Function

2.10.4.1. The AKG will form search keys from a combination of keys for each regularized segment in the Surname field and the regularized GN1.

2.10.4.2. Initials

All names that contain the same first character as the initial will qualify for retrieval on an initial.

2.10.4.3. FNU

All GN1 names qualify for retrieval with a GN1 of FNU (First Name Unknown).

2.10.4.4. Search Keys

2.10.4.4.1. The AKG will generate a set of Search Keys for each input name by conjoining each GN1 key with each SN key of the regularized, repositioned input name.

2.10.4.4.2. All search keys generated for an add will be stored with the record add and associated with the regularized, repositioned form of the name.

2.10.4.5. Generating Keys

2.10.4.6. The AKG will produce two categories of keys:

1. Single-Part Key (SI): a key formed from the single name segment (SN or GN1). All Single-Part Keys will be used to form the Search Keys. There are three kinds of Single-Part Key:
 - Primary Key (PK): a key formed on a single name segment (SN or GN1) and used to define the set of keys for that segment;
 - Wild-Card Key (WK): a key based on the Primary Key that contains wild-card characters;
 - Special Key (SP): a key formed on a single name segment and intended to handle specific variation in the regularized name.

2. Search Key (SK): a multipart key that will be stored and used for retrieval, consisting of a combination of the keys associated with every SN segment and those associated with the GN1.

2.10.4.7. Single-Part Key (SI)

1. The Primary Key (PK)

- The PK is formed from one name segment.
- The PK has a maximum of three characters.
- The PK has the form CCC or CC or C, where C represents any consonant (except in the leftmost position where C may be a vowel).
- The PK is formed from the leftmost character (vowel or consonant) of the regularized segment and the following two consonants (including H, Y and W). If fewer than two additional consonants are available, then the PK may be shorter.

2. The Wild-Card Key (WK)

- The WK is formed from the Primary Key.
- The WKs will have the forms *CC, C*C, CC*, where * represents any consonant, except in the leftmost position where it may represent a vowel.
- The WK will have the forms C* and *C with segments that have only two candidate characters.
- A WK will *not* be formed from a Primary Key with only 1 component (i.e., C).

Figure 4: Example: Formation of Primary and Wild-Card Keys

SEGMENT	PRIMARY KEY	WILD-CARD KEYS		
GAMILA	GML	*ML	G*L	GM*
ABASI	ABS	*BS	A*S	AB*
SAID	SD	*D	S*	
DAI	D	none		

2.10.4.7.1. Special Key (SP)

2.10.4.7.2. The AKG will produce Special Keys (SP) to accommodate situations that cannot be accommodate by the ARR.

2.10.4.7.3. The AKG will generate the Special Keys in addition to the Primary and Wild-Card Keys.

2.10.4.7.3.1. K-Key

2.10.4.7.3.2. The character K alternates with null in many Arabic names, resulting in the potential overlap of many names with the K names.

2.10.4.7.3.3. This phenomenon is not readily handled by the ARR, so names with a K require a Special Key.

2.10.4.7.3.4. The K-Keys are formed in the following way:

2.10.4.7.3.4.1. For any segment with K in initial position, the following keys are produced:
*CC where * represents any character or nothing. (This key is equivalent to a WK produced for this name.)

2.10.4.7.3.4.2. For any segment with K in medial position, the following keys are produced:

1. CkC, where k represents the character "k";
2. CCC, where k has been deleted from the name string and the CCC represents the three leftmost consonants that remain; and

2.10.4.7.3.4.3. For any segment with K in final position, the following keys are produced:

1. CCK, where k represents the character "k" and
2. CC, where k has been deleted from the name string and the CC represents the two leftmost consonants (or vowel in first position) that remain.

2.10.4.7.3.5. The standard set of Wild-Card Keys will also be produced from the Primary Key for K-names.

2.10.4.7.3.6. **Record Add/Query:** The AKG will generate and store all K-Keys with the segment.

Figure 5: Example: Formation of K-Keys

NAME SEGMENT / VARIANT	PRIMARY KEY	K - KEYS	WILD-CARD KEYS
KARSCH	KRS		*RS, K*S, KR*
ARSCH	ARS		*RS, A*S, AR*
MUKBEL	MKB	MBL	*KB, M*B, MK*
MUBEL	MBL		*BL, M*L, MB*
FARUK	FRK	FR	*RK, F*K, FR*
FARU	FR	FR	*R, F*

2.10.4.7.3.7. **High Frequency Key (HK)**

2.10.4.7.3.8. The AKG will generate Special Keys for High Frequency segments found in the input name.

2.10.4.7.3.9. The AKG will access the Arabic Name Type (ANT) Data Store to identify HF segments. (See Section 3.5.)

2.10.4.7.3.9.1. The ANT will contain a set of Arabic name types, the most frequently occurring of which will be specified as High Frequency name segments (HI_FREQ = 1 (True)). (See Section 3.5 for details).

2.10.4.7.3.9.2. The AKG will tag as HF all name segments in the input record that match one of the ARABIC_NAME_TYPE segments for which HI_FREQ = 1 (is True).

2.10.4.7.3.9.3. The AKG will tag all other name segments as LF.

2.10.4.7.3.10. Record Add/Query

2.10.4.7.3.11. The AKG will generate and store the Primary Key for any segment that has been tagged as a HF name segment.

2.10.4.7.3.12. Record Add

2.10.4.7.3.13. The AKG will generate and store all appropriate Wild-Card Keys for any segment that has been tagged as a HF name segment.

Figure 6: Example: Primary Key as HF Key

HF SEGMENT	PRIMARY KEY	WILD-CARD KEYS
MUHAMAD	MHM	*HM, M*M, MH*
AHMED	AHM	*HM, A*M, AH*
ALI	AL	*L, A*

2.10.4.7.4. Search Keys (SK)

2.10.4.7.5. The Search Key is a multipart key formed from all keys associated with one SN segment and all keys associated with the GN1: e.g., *CC + *CC, *CC + C*C, C*C + CC*, etc.

2.10.4.7.6. The Search Keys will be the keys used for retrieval of records from the database.

2.10.4.7.7. Search Key Formation

2.10.4.7.8. To form the set of search keys that will be related to each input record, the AKG will combine each SN segment with the GN1 segment: SN1 + GN1, SN2 + GN1, etc.

2.10.4.7.9. The AKG will determine the frequency (HF or LF) of each of the conjoined segments.

2.10.4.7.10. The number and type of Search Keys will be based on the frequency of the name segments.

2.10.4.7.11. The AKG will form

- Standard Search Keys and
- HF Search Keys.

2.10.4.7.12. **Standard Search Keys(SS)**

2.10.4.7.13. Standard Search Keys (SS) are formed for each SN and GN1 pair.

2.10.4.7.14. **Record Add**

2.10.4.7.15. To form a set of Standard Search Keys, the AKG will combine each Wild-Card Key and each K-Key of each SN segment with each Wild-Card Key and each K-Key of the GN1.

2.10.4.7.15.1. For example, each segment with three characters (CCC) will have generated three Wild-Card Keys.

2.10.4.7.15.2. When the keys from two segments with three characters each are paired, there will be a total of nine keys.

2.10.4.7.15.3. For segments with fewer characters, there will be fewer than nine keys.

2.10.4.7.16. The AKG will generate and store these keys with the record.

Figure 7: Example: Formation of Standard Search Keys (Record Add)

REPOSITIONED, REGULARIZED INPUT FORMAT: AHMED BADAWI, MUHAMAD		
	GN1: MUHAMAD	STANDARD SEARCH KEYS
SN1: AHMED	SN1+GN1: AHMED MUHAMAD	*HM*HM, *HMM*M, *HMMH*, A*M*HM, A*MM*M, A*MMH*, AH**HM, AH*M*M, AH*MH*
SN2: BADAWI	SN2+GN1: BADAWI MUHAMAD	*DW*HM, *DWM*M, *DWMH*, B*W*HM, B*WM*M, B*WMH*, BD**HM, BD*M*M, BD*MH*

2.10.4.7.17. **Query**

2.10.4.7.18. If either segment of the SN + GN1 pair has been tagged as LF, the AKG will generate the Standard Search Keys.

2.10.4.7.19. To form a set of Standard Search Keys, the AKG will combine each Wild-Card Key and K-Key of each SN segment with each Wild-Card Key and K-Key of the GN1. (See Section 2.10.4.7.15)

2.10.4.7.20. **HF Search Keys (HS)**

2.10.4.7.21. **Query**

2.10.4.7.22. If both segments (the SN and the GN1) of the conjoined pair have been tagged as HF segments, the AKG will form *one* Search

Key from the Primary Key of the SN + the Primary Key of the GN1.

2.10.4.7.23. The High Frequency Search Key will be the *only* Search Key used for a *query* on the SN + GN1 pair when both segments are HF segments.

2.10.4.7.24. **Record Add**

2.10.4.7.25. If both segments (the SN and the GN1) that have been conjoined have been tagged as HF segments, the AKG will form *one* Search Key from the Primary Key of the SN + the Primary Key of the GN1.

2.10.4.7.26. The HS will be stored with a record add.

2.10.4.7.27. The HS will be a key stored in addition to the Standard Search Keys for the record.

Figure 8: Example: HF Search Keys

REPOSITIONED, REGULARIZED INPUT FORMAT: AHMED ALI, MUHAMAD		
	GN1: MUHAMAD (HF)	HF SEARCH KEYS
SN1: AHMED (HF)	SN1+GN1: AHMED MUHAMAD	AHMMHM
SN2: ALI (HF)	SN2+GN1: ALI MUHAMAD	ALMHM

2.11. Retrieval Function of the Arabic Search Engine (ASE)

2.11.1. The Arabic Search Engine (ASE) will retrieve records from the database based on the following criteria:

- An exact match of the query Search Keys and stored Search Keys and
- Refusal Code Level and associated Year-of-Birth Range.

2.11.2. The ASE will access the Refusal Code Level/Year-of-Birth Range (RLYOB) Data Store to determine the YOB range within each Refusal Level to search for candidate records.

2.11.3. The ASE will retrieve the unique ID and the **regularized, repositioned** form of the record.

2.11.3.1. Determination of the proximity by the Arabic Filter and Sorter of the query and database records will be based on the regularized, repositioned form of the record.

2.11.3.2. The ASE will eliminate all records with the same unique ID retrieved during the retrieval process.

2.11.4. Subordinates

Arabic Key Generator.

2.12. ARABIC FILTER AND SORTER MODULE DECOMPOSITION (AFS).

2.12.1. Identification

This module is known as the Arabic Filter and Sorter (AFS).

2.12.2. Type

2.12.2.1. The AFS is a module that accepts each regularized database record retrieved by the ASE and compares it to the regularized form of the query record.

2.12.2.2. The AFS is constituted of two subordinate functions:

- the Arabic Filter and
- the Arabic Sorter.

2.12.2.3. The AFS must follow the Arabic Search Engine (ASE).

2.12.3. Purpose

2.12.3.1. The set of database records that the ASE will retrieve will have no value relative to the query record. The AFS will evaluate each of the records retrieved for its proximity to a query record, will retain those that pass a pre-established threshold and will sort the resultant candidate list.

2.12.3.2. The filtering process will take into account a number of factors that play a role in determining the relative value of Arabic names.

2.12.4. Function

2.12.4.1. The AFS will compare the query name and record name to determine a relative surname value and given name value and will generate a composite score for the records by accounting for Date-of-Birth, Refusal Level and Country-of-Birth proximity.

2.12.4.2. Arabic Filter Function of the AFS

2.12.4.3. The Arabic Filter and Sorter will first determine if the query record and prime database record (unregularized version) match exactly.

2.12.4.3.1. The Surname, Given Name, Date-of-Birth and Country-of-Birth must be exact matches.

2.12.4.3.2. If the two records match exactly, the AFS will tag the record as an exact match.

2.12.4.3.3. The AFS will send the record directly to the Arabic Sorter Function.

2.12.4.4. The Arabic Filter and Sorter (AFS) will accept the regularized, repositioned candidate records retrieved by the ASE.

2.12.4.5. The AFS will perform a digraph comparison of the regularized, repositioned surname segments (stems) of the query record and each candidate record.

2.12.4.6. The AFS will perform a digraph comparison of the regularized given name segment (stem) of the query record (GN1) and the given name segment (stem) of each candidate record (GN1).

2.12.4.6.1. The score produced by the digraph comparison (DI_VAL) will be adjusted by values assigned to several parameters.

2.12.4.6.2. The score assigned to the surname and to the given name, after the parameters have adjusted the DI_VAL, will be the SN_VAL and the GN_VAL.

2.12.4.6.3. Factors that contribute to the determination of the name scores (SN_VAL and GN_VAL) include

- SNTHR
- GNTHR
- OPVAL
- INITSN
- INITGN
- TAQASN
- TAQAGN
- TAQXSN
- TAQXGN
- GNDR

2.12.4.6.4. A final name score will be calculated for each candidate database record as it compares to the query record.

2.12.4.6.4.1. A score for the SN will be calculated: SN_VAL.

2.12.4.6.4.2. A score for the GN will be calculated: GN_VAL.

2.12.4.6.5. To be included in the final candidate list, the SN_VAL and GN_VAL must each pass pre-determined SN and GN threshold levels (SNTHR and GNTHR).

2.12.4.7. Surname Evaluation

2.12.4.8. The AFS will perform a digraph comparison of each SN stem of the database record with each SN stem of the query record.

2.12.4.8.1. The digraph value is determined in the following way:

2.12.4.8.1.1. The digraphs are identified for each name stem.

2.12.4.8.1.1.1. Each pair of alphabetic characters is identified: TAFIQ → TA / AF / FI / IQ

2.12.4.8.1.1.2. A digraph is also formed of the initial boundary (#) and the first alphabetic character:
TAFIQ → #T.

2.12.4.8.1.1.3. A digraph is also formed of the final alphabetic character and the final boundary (#):
TAFIQ → Q#.

2.12.4.8.1.2. The number of shared digraphs is calculated.

2.12.4.8.1.2.1. A digraph may match one digraph only.

2.12.4.8.1.3. The number of shared digraphs is multiplied by 2 and divided by the total number of digraphs in Comparand #1 added to the total number of digraphs in Comparand #2.

2.12.4.8.1.3.1. The formula is:

$2 * d / a + b$,
where d = the total number of shared digraphs;
where a = the total number of digraphs in Comparand #1; and
where b = the total number of digraphs in Comparand #2.

2.12.4.8.1.4. The result is the Digraph Value (DI_VAL) for the two Comparands.

Figure 9: Example: Digraph Calculation

COMPARANDS	DIGRAPHS	SHARED DIGRAPHS	DI_VAL
COMPARAND #1: BADIR	#B BA AD DI IR R# (6 total digraphs = a)	BA AD DI IR R#	$2 * d / a + b = 10 / 13$
COMPARAND #2: ABADIR	#A AB BA AD DI IR R# (7 total digraphs = b)	= 5 (d)	0.77

2.12.4.9. This process is performed for each of pair of Comparands in the database and query SN (SN1/SN1, SN1/SN2, SN1/SN3, SN2/SN2, etc.).

2.12.4.10. Each DI_VAL is adjusted according to parameter values in the Filter Parameter Data Store (see Section 3.6 for details).

2.12.4.11. The AFS will determine if the appropriate parameter conditions are met.

2.12.4.12. If the appropriate conditions are present, the DI_VAL will be multiplied by the value assigned to the parameter and the relative score of the two Comparands will be lowered.

2.12.4.13. Parameter Conditions

2.12.4.13.1. INITSN: Surname Initial

2.12.4.13.1.1. Definition: A SN segment is a single character and it matches the first character of the other comparand.

2.12.4.13.1.2. Action: Assign the INITSN value to the comparison value (i.e., do not calculate the DI_VAL).

2.12.4.13.2. OPSN: Out-of-Place Surname

2.12.4.13.2.1. Definition: A SN segment that is not in the same relative position in the SN string in both the database and query records.

2.12.4.13.2.2. Action: Multiply the DI_VAL by the OPSN value. (See Figures 10 and 11.)

2.12.4.13.3. TAQ Filter

2.12.4.13.4. All TAQ tags (ID_NO, disposition, TAQ_TYPE and associated SN stem) will be retrieved with the database record.

2.12.4.13.5. The AFS will evaluate any TAQs associated with the SN segments being evaluated, except Stranded Affixes (see Section 2.5.4.2.7.3).

2.12.4.13.5.1. A Stranded Affix will not play a role in the prefix comparison.

2.12.4.13.6. Single TAQs

2.12.4.13.7. Missing TAQs

2.12.4.13.8. TAQASN: Absent TAQ Value

2.12.4.13.8.1. Definition 1: One of the two comparands has a TAQ tag, the other does not.

2.12.4.13.8.2. Definition 2: Both SN segments have a single TAQ tag, one is a TAQ DELETE, the other a TAQ DISREGARD.

2.12.4.13.8.3. Action: Multiply the DI_VAL by the TAQASN value. (See Figures 12 and 22.)

2.12.4.13.9. TAQ DELETE

2.12.4.13.9.1. If the TAQ DELETE tags refer to the same TAQ segment, the DI_VAL will be unchanged.

2.12.4.13.9.2. If the TAQ DELETE tags refer to different TAQ DELETE segments, multiply the DI_VAL by the TAQXSN value. (See Figure 22.)

2.12.4.13.10. TAQ DISREGARD Processing

2.12.4.13.10.1. The AFS will access the TAQ Filter Data Store (TF) to process SN TAQ segments that have been tagged as DISREGARD.

2.12.4.13.10.2. Definition: The AFS will access the TAQ Filter Data Store (TF) to process records if they both contain SN TAQ segments that have been tagged as DISREGARD.

2.12.4.13.10.3. Action 1: The AFS will assign TAQDIS#1 to the TAQ DISREGARD segment for the database SN segment and TAQDIS#2 to the TAQ DISREGARD segment for the query SN segment.

2.12.4.13.10.4. Action 2: If the two TAQ DISREGARD segments match, the DI_VAL will remain unchanged.

2.12.4.13.10.5. Action 3: If the two TAQ DISREGARD segments do not match, the AFS will identify the TF_VALUE for the pair in the TF. (See Figure 24.)

2.12.4.13.10.5.1. The AFS will multiply the DI_VAL by the TF_VALUE for the pair.

2.12.4.13.11. Multipart TAQs

2.12.4.13.11.1. Definition: If at least one SN comparand has multipart TAQ tags (they may be all DISREGARD, all DELETE, or mixed DISREGARD/DELETE), the AFS will perform the following analyses.

2.12.4.13.11.2. Action: If all TAQs match, AFS will make no change in the DI_VAL.

2.12.4.13.11.3. TAQ DELETES

2.12.4.13.11.3.1. Definition: All DELETE tags

2.12.4.13.11.3.2. Action 1: If any DELETE TAQ matches, the AFS applies no change.

2.12.4.13.11.3.3. Action 2: If no DELETE TAQs match, multiply the DI_VAL by the TAQXSN Value.

2.12.4.13.11.4. TAQ DISREGARDs

2.12.4.13.11.4.1. Definition: All DISREGARD tags

2.12.4.13.11.4.2. Action 1: If any TAQ DISREGARD segment matches, the AFS will make no change in the DI_VAL.

2.12.4.13.11.4.3. Action 2: If no TAQ DISREGARD segments match, the AFS will identify the highest match value from the TF (TF_VALUE) and multiply that by the DI_VAL. (See Figures 23 and 24.)

2.12.4.13.11.5. TAQ DISREGARD and DELETES

2.12.4.13.11.5.1. Definition: Mixed DISREGARD/DELETE tags

2.12.4.13.11.5.2. Action 1: If DISREGARD segments are present in both comparands and there is any match among the DISREGARD segments, the AFS will make no change in the DI_VAL.

2.12.4.13.11.5.3. Action 2: If DISREGARD segments are present in both comparands and there is no match among the DISREGARD segments, the AFS will determine the highest match value from the TF for any DISREGARD tags and multiply the DI_VAL by that value. (That is, ignore any DELETE tags.)

2.12.4.13.11.5.4. Action 3: If a DISREGARD segment is in one comparand and not the other and the two comparands have at least one DELETE tag that matches, the AFS will make no change in the DI_VAL.

2.12.4.13.11.5.5. Action 4: If a DISREGARD segment is in one comparand and not the other and the two comparands have DELETE tags that do not match, multiply the DI_VAL by the TAQXSN. (See Figure 22.)

2.12.4.14. After all evaluations have been performed, the AFS will choose the highest score for each name segment.

2.12.4.14.1. The highest score for both the row and column must be chosen.

2.12.4.14.2. Only one score per row and column is permitted.

2.12.4.14.3. If two scores are equal, only one is chosen.

Figure 10: Example 1: Digraph Evaluation: Equal Number of SN Segments; Digraph Variants BADAWI/BEDAWI

	AHMED	ALI	BADAWI
AHMED	1.00	0.20	0.00
ALI	0.20	1.00	0.18
BEDAWI	0.15	0.18	0.71

Figure 11: Example 2: Digraph Evaluation: Different Number of SN Segments; OPSN applies to BADAWI/BEDAWI

	AHMED	ALI	BADAWI
AHMED	1.00	0.20	0.00
BEDAWI	0.15	0.18	0.61

Figure 12: Example 3: Digraph Evaluation: Same Number of SN Segments; TAQ Tag Present on One SN

	(ABU) AHMED	SALIM	SAYED
AHMED	0.90	0.00	0.28
SAID	0.00	0.36	0.47
AKBAR	0.16	0.00	0.00

Figure 13: Example 4: Digraph Evaluation: Same Number of SN Segments; Different TAQ_DISREGARD Segments Present

	(ABU) AHMED	SALIM	SAYED
(BIN) AHMED	0.50	0.00	0.28
SAID	0.00	0.36	0.47
AKBAR	0.16	0.00	0.00

2.12.4.15. The AFS will sum the values chosen from the comparison matrix and will divide by the number of values chosen to produce the SN_VAL.

2.12.4.15.1. In Example 1, $1.00 + 1.00 + 0.61/3 = 0.87$

2.12.4.15.2. In Example 2, $1.00 + 0.61/2 = 0.81$

2.12.4.15.3. In Example 3, $0.90 + 0.47 + 0.00/3 = 0.46$

2.12.4.15.4. In Example 4, $0.50 + 0.47 + 0.00/3 = 0.32$

2.12.4.16. The AFS will compare the SN_VAL to the SNTHR.

2.12.4.16.1. The SN_VAL must be equal to or greater than the SNTHR.

2.12.4.16.2. The record must pass the SNTHR to qualify for the final candidate list.

2.12.4.17. Given Name Evaluation

2.12.4.18. The GN has only one segment, the GN1.

2.12.4.18.1. The AFS will perform a digraph comparison on the regularized GN1 stem of the database record and the regularized GN1 of the query record.

2.12.4.18.2. The DI_VAL will be calculated as it was for the SN (see Section 2.12.4.8).

2.12.4.18.3. The DI_VAL will be adjusted by several GN parameters.

2.12.4.18.4. **INITGN**: Given Name Initial

2.12.4.18.4.1. Definition: A GN1 is a single character and matches the first character of the GN1 of the other comparand.

2.12.4.18.4.2. Action: Assign the INITGN value to the comparison value (i.e., do not calculate a DI_VAL)

2.12.4.18.5. **TAQ Evaluation** will proceed as with the SN, mutatis mutandi (see Section 2.12.4.13.3).

2.12.4.18.6. **GNDR**: Record Gender Value

2.12.4.18.6.1. The AFS will compare the record gender of the input name and the query name.

2.12.4.18.6.2. If the genders match, no action will take place.

2.12.4.18.6.3. If the genders do *not* match, multiply the DI_VAL of the GN1 by the GNDR value. (See Figure 24.)

2.12.4.19. The value resulting from all GN1 calculations will be the GN_VAL.

2.12.4.20. The AFS will compare the GN_VAL to the GNTHR. (See Figure 24.)

2.12.4.20.1. The GN_VAL must be equal to or greater than the GNTHR.

2.12.4.20.2. The record must pass the GNTHR to qualify for the final candidate list.

2.12.4.21. Composite Score

2.12.4.22. The AFS will develop a Composite Score for the two comparands.

2.12.4.23. The AFS will adjust the GN_VAL and the SN_VAL by factors that reflect the proximity of the Refusal Level, Date of Birth and Country of Birth.

2.12.4.24. The GN_VAL and SN_VAL will be multiplied by factors that apply to the RL, DOB and COB.

2.12.4.25. Refusal Level Factor

2.12.4.26. The AFS will access the Refusal Code Level Data Store to determine the Refusal Level Category of the Refusal Code.

2.12.4.27. The AFS will access the Filter Parameter Data Store to find the PARM_VAL associated with the Refusal Level (RL#).

2.12.4.28. Date-of-Birth Factor

2.12.4.29. The AFS will access the Year-of-Birth Range Data Store to determine the YOB Category, YOB#, of the Dates-of-Birth of the comparands. The highest value is applied to the relationship.

2.12.4.30. The AFS will access the Filter Parameter Data Store to find the PARM_VAL associated with the YOB Category (YOB#).

2.12.4.31. Country-of-Birth Factor

2.12.4.32. The AFS will access the Country of Birth Category Data Store to determine the COB Category, COB#.

2.12.4.33. The AFS will access the Filter Parameter Data Store to find the PARM_VAL associated with the Country of Birth Category (COB#).

2.12.4.34. The AFS will calculate a composite score by multiplying the SN_VAL by the GN_VAL by the RL# PARM_VAL by the YOB# PARM_VAL by the COB# PARM_VAL.

2.12.4.35. Final Sort Function of the AFS

2.12.4.36. The AFS will rank order the final candidate list of database records.

2.12.4.37. The prime (unregularized) record will be returned to the user.

2.12.4.37.1. There may be significant differences between the query record and the qualifying database records.

2.12.4.37.2. The Composite Score will be returned with the record.

2.12.4.38. Any record that is tagged as an exact match will be placed at the top of the list.

2.12.4.39. All remaining records in descending order of Composite Score.

2.12.4.40. The goal of the final sort is to place exact record matches on the top and to rank order the remaining records by the degree of contribution that each data element (SN, GN, DOB, COB, Refusal Code Level (RL)) makes to the overall record value.

2.12.4.41. The details of the sort will be derived from extensive discussion about the business requirements.

2.12.4.42. Because the scores from the various pipes may not have been calculated in the same way, a method for evaluating the relative value of candidate records will have to be devised.

2.12.4.43. Internal Order

2.12.4.43.1. There may be cases in which the sorting criteria are met equally by more than 1 record.

2.12.4.43.2. Where multiple records qualify equally, there will be an internal sort order.

2.12.4.43.2.1. SN Score

2.12.4.43.2.2. GN Score

2.12.4.43.2.3. DOB Levels

2.12.4.43.2.4. Refusal Levels

2.12.4.43.2.5. COB Relationships

2.12.4.44. The AFS will return the top n records to the central CLASS-E sorter.

2.12.4.44.1. The number of records to be returned will be a system setting.

2.13. LINGUISTIC TRACE FACILITY MODULE DECOMPOSITION

2.13.1. Identification

This module is known as the Linguistic Trace Facility (LTF).

2.13.2. Type

The LTF is a program that will interact with any or all modules and functions within those modules.

2.13.3. Purpose

The LTF will allow system evaluators to access information about the system functions so that the quality of the content can be ensured. To diagnose and remedy problems associated with questionable system results, evaluators must

have access to the results of system functionality at various points during the processing cycle.

2.13.4. Function

2.13.4.1. The LTF will be a mechanism that will copy and divert statistics, information, processing results to a file outside the main processing module.

2.13.4.2. The file will be readily accessible on-line for examining by a system evaluator.

2.13.4.3. Multiple trace points will be identified when the system is built.

2.13.4.4. Examples of trace points:

- What ARR's (by ID_NO) have applied
- Regularized, repositioned name form
- All keys generated for a query and for an add
- SN and GN DI_VAL
- SN_VAL and GN_VAL
- Record Gender
- Sort considerations

3. DATA DECOMPOSITION

3.1. DATA

3.1.1. The input data for an ANA-E query will contain all information that is currently required by CLASS and in the standard format required by CLASS.

- NAME (Surname, Given Name);
- DOB (Date of Birth; Day Month Year); and
- COB (Country of Birth; FIPS codes).

In addition, the following will be specified:

- Applicant Gender (AG): Male (M), Female (F), Unknown (U).
- A unique identifier (UID) (as defined in CLASS-E).

3.1.2. For adds, other record information will be entered, as required by CLASS and CLASS-E: e.g., refusal code, province of birth.

3.2. DATA STORES

The following data stores will be accessed by the ANA-E processing components:

- Arabic Regularization Rules Data Store (ARR)
- Arabic Title/Affix/Qualifier Data Store (ATD)
- Arabic Name Type Data Store (ANT)
- Filter Parameter Data Store (FP)
- TAQ Filter Data Store (TF)
- Refusal Code Level Data Store (RCL)

- YOB Range Data Store (YR)
- Refusal Code Level/Year-of-Birth Range Data Store (RLYOB)
- COBPROX Data Store (COBPROX)
- Arabic COB Category Data Store (ACOB)

3.3. ARABIC REGULARIZATION RULES DATA STORE DECOMPOSITION

3.3.1. Identification

This rule base is known as the Arabic Regularization Rule Base (ARR).

3.3.2. Type

3.3.2.1. The ARR is a set of transformation rules accessed by the Arabic Rule Engine.

3.3.2.2. The ARR will have the following format:

Figure 14: Format: Arabic Regularization Rule Base

FIELD NAME	DATA TYPE	FIELD SIZE	DATA VALUE
ID_NO	integer	3	001...999
PRE-CONTEXT	character	unlimited	any ASCII character
IN	character	unlimited	any ASCII character
POST-CONTEXT	character	unlimited	any ASCII character
OUT	character	unlimited	any ASCII character

3.3.2.3. Definitions

3.3.2.3.1. ID_NO: a unique, arbitrary numerical reference to the rule.

3.3.2.3.2. PRE-CONTEXT: preceding context for the element to be matched; delimited by preceding and following quotation marks (" ")

3.3.2.3.3. IN: the match context; the portion of the name that will undergo change; delimited by preceding and following quotation marks (" ")

3.3.2.3.4. POST-CONTEXT: trailing context for the element to be matched; delimited by preceding and following quotation marks (" ")

3.3.2.3.5. OUT: the rule output; the realized change in the IN; delimited by preceding and following quotation marks (" ")

3.3.2.4. There is no internal limit on the size of the Pre-Context, In, Post-Context or Out, although the system may have an external limit (e.g., the maximum size of the SN field).

3.3.2.5. All rules will use standard regular expression notation, with one exception (\$), which has been defined specifically for this rule base.

3.3.2.6. Regular Expression Notation

Figure 15: Regular Expression Notation

REGEXP. NOTATION	DEFINITION
.	Matches any single character, including white space.
-	Stands for all characters that come between the two characters given. This is a standard "from-to" notation; with characters, it presumes an A to Z character set. For example, [A-D] will match on A or B or C or D. (See [] below.)
[]	Identifies a class of characters; a match can occur on a single occurrence of any single element within the []: [OU] will match O or U. For example, "J[OU]N" will match on JON or JUN but not on JOUN. If a + is added to the bracketed expression, "J[OU]+N", it will match on any combination of any number of Os and Us: JOUUN, JUUON, JOUOUN, JUUN, JOUUN, etc. In contrast, OU without [] will match only on the exact combination of characters OU: "JOUN" will match on JOUN only.
+	Matches one or more occurrences of a preceding character or regular expression, in any order. For example, JO+N will match JON or JOON or JOOON, etc., [OU]+ matches OOOU or OUOUOU or O or UO or OOU or UUUO, etc.
?	Matches zero or <i>one</i> occurrence of the preceding regular expression. The expression "JOH?N" will match on JON or JOHN but will not match on JOHNN.
*	Matches zero or more occurrences of the preceding regular expression. The expression "JOH*N" will match on JON, JOHN, JOHNN, JOHHHNN, etc.
()	Groups together regular expressions. "(J[OU]N)H(AE)RRY)" will match on JON or JUN or HARRY or HERRY. (Contrast with [] which identifies a character class and contrast with { } which identifies a metasympol.)
→ ()	Matches <i>either</i> the preceding regular expression <i>or</i> the following regular expression. The full expression is all contained within (). For example, (AB AP) will match the character string AB or AP. The same expression may also be written A[BP].
" "	Defines the context boundary: the Preceding Context, Match Context, Post Context and Output. A context that is made up of only one metasympol and is not bracketed by () should not be surrounded by " ". For example, the metasympol Consonant can stand alone. If the metasympol is enclosed within { }, then all regular expressions contained within the context must be enclosed within " ". For example, "{Consonant}{Letter}" within one context requires both { } and " ".
{ }	Contains one or more pre-defined metasympols. If { } are used, they must be surrounded by " ". If a single metasympol occurs alone, no { } are necessary and therefore no " " are necessary. For example, the single metasympol Vowel can appear either as Vowel or "{Vowel}". If more than one element is used, the metasympol must all appear within { } and the whole string within " ". E.g., Vowel, "{Vowel}", "J{Vowel}HN" are acceptable formats.
\$	Indicates the character to output. \$ is defined differently from other standard definitions. \$ is a variable that is followed by an integer that references a character in the match string. For example, each character in an input string is associated with a different, consecutive integer value, up to the number of characters in the match: JONES becomes J = \$1, O = \$2, N = \$3, E = \$4, S = \$5. Reference can be made to the index values in the output string. \$1 \$2 \$2 \$3 \$3 \$5 would represent JOONNS.

3.3.2.7. Metasymbols

A number of meta-symbols will be accessed by the rules. The metasymbols are variables declared at the beginning of the ARR Data Store.

Figure 16: ARR Metasymbols

METASYMBOL	DEFINITION
Letter	"[A-Z]"
Consonant	"[BCDFGHJKLMNPQRSTVWXYZ]" (N.B. Includes W and Y)
Nog (= No Glide)	"[BCDFGHJKLMNPQRSTXZ]" (N.B. No W or Y)
Alla	"[AEIOU]+L+[EA]+H?"
Rhyme	"[AEIOUY]+[BCDFGHJKLMNPQRSTVWXYZ]"
Kesra	"([EA]?[IY]+[IE]+Y)"
Dad	"(Z+ TH+ DH+ DD+)"
Tha	"(Z+ TH+ DH+ C+ S+ T+)"
Jim	"(DJ J Y G DZH DSCH GG DY)"
Gine	"(G GH RH)"
Qaf	"(Q G K J KH GH C QU CK)"
Kha	"(KH K X Q C)"
Marbuta	"([EAI]H [AE]T?)"
Sun	"(C S N D T R Z G J)"
Sungem	"(SS NN DD TT RR ZZ GG JJ)"
Moongem	"(BB FF GG HH JJ KK MM NN PP QQ VV WW XX)"
Vowelgem	"(AA EE II OO UU)"
Vowel	"[AEIOU]"
Didi	"(KHKH SHSH GHGH RHRH DHDH THTH CHCH PHPH)"
Dig	"[KSGRDTCP]H"
Bound	" " (= white space)
Anything	" "
Othergem	"(CC LL YY)"

3.3.2.8. Purpose

The ARR allow records with highly divergent spellings and/or representations of the same name to be retrieved from the database. Usual character comparison techniques are unable to retrieve records with these variants.

3.3.2.9. Function

The ARR applies relevant rules to each Arabic name field and produces a common representation for variant realizations of the same name.

MUHAMMED, MOHAMMAD and IMHEMED are variant forms of the same name; each will be set equal to one single representation of the name: **MUHAMAD**, for example. The successful application of one or more rules will produce as output a regularized Arabic name string.

3.3.2.10. Examples

3.3.2.10.1. Example 1 contains two rules that apply to variants of **ABDULLA**:

- EVDILLAH
- ABD ALA
- ABDU ALLA
- ABDULLAH
- OABDELA
- AABDILA
- ABDELILA

Figure 17: Example 1: "ABDULLA" Regularization Rules

ID _NO	PRE- CONTEXT	IN (MATCH CONTEXT)	POST- CONTEXT	OUT
676	Bound	"[HKCQ]?[AE]+[BV]*D+{Vowel}?{Bound}{Alla}"	Bound	"abdula"
677	Bound	"[HKCQ]?[AE]+[BV]*D+[AEIOU]*L+[IE]*L*AH?"	Bound	"abdula"

3.3.2.10.2. Example 2 contains one rule that applies to variants of G:

- MAGUID
- MADZHID
- MADSCHID
- MADJID
- MAJID
- GHASSAN

Figure 18: Example 2: "G" Regularization Rules

ID _NO	PRE- CONTEXT	IN (MATCH CONTEXT)	POST-CONTEXT	OUT
132	Anything	"(D J GH DSCH DZH J+)"	Anything	"g"

3.4. ARABIC TITLE/AFFIX/QUALIFIER DATA STORE DECOMPOSITION

Because the ANA-E design is viewed as an independent sub-program of the CLASS-E system, the Arabic Title/Affix/Qualifier Data Store is presented here as a separate table. It is strongly suggested, however, that CLASS-E support one TAQ Data Store in which the cultural affinity of each TAQ segment is indicated. This is reduce table maintenance and will provide a global picture of the handling of TAQs.

3.4.1. Identification

This data store is known as the Arabic Title/Affix/Qualifier Data Store (ATD).

3.4.2. Type

The ATD is a data store that contains the Arabic-specific Title, Affix and Qualifier segments and their distribution. It will be accessed by the Arabic Preprocessor (APP) and the Arabic Filter and Sorter.

Figure 19: Format: Arabic TAQ Data Store

DATA FIELD	DATA TYPE	FIELD SIZE	DATA VALUE
ID_NO	integer	4	1...9999
TAQ FORM	character	15	alphabetics
TAQ TYPE	character	1	T, P, I, S, Q
DELETE	integer	1	1, 0 (True, False)
DISREGARD	integer	1	1, 0 (True, False)

3.4.2.1. Definitions

3.4.2.1.1. ID_NO: a unique, arbitrary number that identifies the TAQ segment.

3.4.2.1.2. TAQ FORM: the string that represents the TAQ; the TAQ FORM may be a multipart string (i.e., a string that includes internal white space).

3.4.2.1.3. TAQ TYPE: an indicator of the kind of TAQ segment present: a title (T), prefix (P), infix (I), suffix (S) or qualifier (Q).

3.4.2.1.4. DELETE:

3.4.2.1.4.1. The segment is to be removed from all further consideration in the name search process; it will contribute marginally to the filtering process. It will be returned with the record to the user.

3.4.2.1.4.2. The segment is referenced in the filtering process.

3.4.2.1.4.3. The segment is not removed from the original record and is returned with the record to the user.

3.4.2.1.4.4. True (1) or False (0) indicates whether or not this function is to apply to the segment(s) under consideration.

3.4.2.1.5. DISREGARD:

3.4.2.1.5.1. The segment is to be removed from further consideration in the name search process but will undergo special evaluation in the filtering process. It will be returned with the record to the user.

3.4.2.1.5.2. True (1) or False (0) indicates whether or not this function is to apply to the segment(s) under consideration.

3.4.3. Purpose

Peripheral elements (Titles, Affixes, and Qualifiers) in names do not contribute as much to the name evaluation as does the name stem. Identifying and

removing these elements in the name processing component is important. They do, however, contribute to the overall value of a name when compared to another name. They will therefore contribute some value to the filtering and sorting processes.

3.4.4. Function

The ATD serves as a repository for all TAQ values and for the treatment that each will be subjected to.

3.5. ARABIC NAME TYPE DATA STORE DECOMPOSITION

3.5.1. Identification

This data store is known as the Arabic Name Type Data Store (ANT).

3.5.2. Type

3.5.2.1. The ANT is a data store of unique regularized Arabic name segments.

3.5.2.2. The ANT is generated only after regularization has applied to the input name.

3.5.2.3. The ANT will have the following format:

Figure 20: Format: Arabic Name Type Data Store

DATA FIELD	DATA TYPE	FIELD SIZE	VALUE RANGE
ID_NO	integer	5	00001...99999
ARABIC_NAME_TYPE	character	24	alphabetics
GENDER	character	1	M, F, U
HI_FREQ	integer	1	1, 0 (True or False)

3.5.2.4. Definitions

3.5.2.5. ID_NO: a unique, arbitrary numerical reference to the name segment (ARABIC_NAME_TYPE)

3.5.2.6. ARABIC_NAME_TYPE: unique entries that correspond to the *regularized* form of a name segment.

3.5.2.7. GENDER: the gender associated with a particular name segment: M (Male), F (Female), U (Unknown/Unspecified). As records are added to the ANT, gender will be specified as U. The gender assigned to new table entries will be periodically reevaluated so that names that can be identified for gender can be appropriately marked.

3.5.2.8. HI_FREQ: the frequency of all names will be indicated. True (1) will indicate that a name segment is considered a high frequency Arabic name segment. All other segments will be marked as False (0), a low-frequency name segment.

3.5.3. Purpose

The purpose of the ANT data store is to reduce the need to perform repeated digraph comparisons on a large store of names and to permit the retrieval of gender-matching records.

3.5.4. Function

The ANT will provide information about the distinct Arabic name types, their frequency and gender.

3.6. FILTER PARAMETER DATA STORE DECOMPOSITION

3.6.1. Identification

This module is known as the Filter Parameter Data Store (FP).

3.6.2. Type

3.6.2.1. The FP is a data store that will be accessed by the Filter Component of the Arabic Filter and Sorter (AFS).

3.6.2.2. The FP is a parameter table that will be accessible to and adjustable by the user and whose cell values will be determined through testing and comparative evaluation.

3.6.2.3. The FP has the following format:

Figure 21: Format: Filter Parameter Data Store

DATA FIELD	DATA TYPE	FIELD SIZE	VALUE RANGE	DATA VALUE
PARM_NAME	character	6	alphabetic	SNTHR, GNTHR, OPSN, INITSN, GNDR, INITGN, TAQASN, TAQAGN, TAQXSN, TAQXGN.

				RL#, YOB#, COB#
PARM_VAL	decimal	4	0.00...1.99	Various (TBD)

Figure 22: Example: Filter Parameter Data Store (Values are for example only.)

PARM_NAME	PARM_VAL
SNTHR	0.60
GNTHR	0.65
OPSN	0.60
INITSN	0.85
INITGN	0.85
GNDR	0.65
TAQASN	0.90
TAQAGN	0.90
TAQXSN	0.85
TAQXGN	0.85
RL0	1.20
RL1	1.15
RL2	1.10
RL3	1.05
RL4	1.00
YOB0	1.30
YOB1	1.25
YOB2	1.20
YOB3	1.15
YOB4	1.10
YOB5	1.05
YOB6	1.00
COB1	1.20
COB2	1.15
COB3	1.10
COB4	1.00
COB5	0.95

3.6.2.4. The values provided are as examples only and do not necessarily represent the PARM_VALs to be used for the parameters.

3.6.3. Purpose

The FP is a data store that allows easy access to adjustable parameters that contribute to the determination of the composite score of two record comparands.

3.6.4. Function

The FP functions as an independent data store with all the adjustable parameters needed by the AFS during the filtering process.

3.7. TAQ FILTER DATA STORE DECOMPOSITION

3.7.1. Identification

This data store is known as the TAQ Filter Data Store (TF).

3.7.2. Type

3.7.2.1. This TF will be accessed by the Arabic Filter and Sorter and provides parameter factors for matching TAQ DISREGARD tags during record filtering.

3.7.2.2. The format of the TF follows:

Figure 23: Format: TF Matrix Design

DATA FIELD	DATA TYPE	FIELD SIZE	VALUE RANGE	DATA VALUE
TAQDIS#1	character	8	alphabetic	TAQ_DISREGARD ITEM
TAQDIS#2	character	8	alphabetic	TAQ_DISREGARD ITEM
TF_VALUE	decimal	4	0.00...1.00	Various (TBD)

3.7.2.3. Definitions

3.7.2.4. TAQDIS#1: is the TAQ DISREGARD segment that occurs in one or the other (different) of the comparands.

3.7.2.5. TAQDIS#2: is the TAQ DISREGARD segment that occurs in one or the other (different) of the comparands.

3.7.2.6. TF_VALUE: is the factor that will be used to adjust the SN_VAL or GN_VAL if the TAQDIS#1 and TAQDIS#2 are present in the comparands.

Figure 24: Example: TF Sample (Values are for example only)

TAQDIS#1	TAQDIS#2	TF_VALUE
ABD EL	ABD EL	1.00
ABD EL	ABU	0.75
ABD EL	AL	0.85
ABD EL	BIN	0.75
ABD EL	EL DIN	0.50
ABU	ABU	1.00
ABU	AL	0.85
ABU	BIN	0.50
ABU	EL DIN	0.85
AL	AL	1.00
AL	BIN	0.85
AL	EL DIN	0.50
BIN	BIN	1.00
BIN	EL DIN	0.85
EL DIN	EL DIN	1.00

3.7.3. Purpose

Arabic names often have peripheral name elements. Some of these make up a segment of the name, the TAQ values identified in the TF. Their relative value, however, varies. Some of them cannot cooccur, some have opposite

meanings, so it is necessary to identify their relative value when they are contrasted with one another.

3.7.4. Function

The TF provides the resources for the AFS to determine the relative value of two TAQs that occur in two comparands.

3.8. REFUSAL-CODE LEVEL DATA STORE DECOMPOSITION

3.8.1. Identification

This data store is known as the Refusal Code Level Data Store (RCL).

3.8.2. Type

3.8.2.1. It is recommended that the RCL be a parameter file, which can be accessed by the client so RC categories can be added to or changed with ease.

3.8.2.2. The RC data store will provide a list of the Refusal Codes and the level of seriousness of each Refusal Code.

3.8.2.3. The RCL has the following format:

Figure 25: Format: Piece of Refusal Code Level Data Store (RCL)

DATA FIELD	DATA TYPE	FIELD SIZE	VALUE	CATEGORY DEFINITION
00	alphanumerics	3	RL0	Most serious RC: 00
23	alphanumerics	3	RL1	Type 1 Serious RCs
6C	alphanumerics	3	RL2	Type 2 Serious RCs
07	alphanumerics	3	RL3	Type 1 Non-serious RCs
G	alphanumerics	3	RL4	Type 2 Non-serious RCs
...				

3.8.2.4. Definitions

3.8.2.4.1. DATA FIELD: indicates each Visa Refusal Code (Codes and their Refusal Level (see VALUE) are for example only; they do not represent the complete list nor the accurate assignment of a Refusal Code to a Refusal Level).

3.8.2.4.2. DATA TYPE: The RL# will appear in the form RL1, RL2, etc.

3.8.2.4.3. VALUE: RL# is the Refusal Level category to which are particular Refusal Code has been assigned. The Visa Office will assign Refusal Codes to one of 4 categories: RL1, RL2, RL3, RL4; RL0 is reserved for the Refusal Code 00. (The current distinction among Refusal Codes is a binary one: serious and non-serious. Assignment of Refusal Codes to more groups has not yet been done; the consequence is that one or more of these categories may not have a distinct value.) The RL# occurs in

ascending order, from most serious to least serious Refusal Code. The RL# will be linked to a Year-of-Birth Code (see Section 3.9) to determine the relevant subsets of records to be searched.

3.8.2.4.4. CATEGORY DEFINITION:

- RC0 refers to the Refusal Code 00.
- RC1 refers to all Refusal Codes that have been designated as Type 1 Serious RC 1, i.e., the most serious, excluding 00.
- RC2 refers to all Refusal Codes that have been designated as Type 2 Serious RC, i.e., serious but less serious than RC0 and RC1.
- RC3 refers to all Refusal Codes that have been designated as Type 1 Non-Serious RC. These codes are less serious than the RC0, RC1 and RC2 codes.
- RC4 refers to Refusal Codes that have been designated as Type 2 Non-Serious. These codes are the least serious codes, less serious than the RC0, RC1, RC2 and RC3 codes.

3.8.3. Purpose

It has long been desirable to make more granular distinctions among the Refusal Codes. For many years, DOS has maintained a distinction between serious and non-serious codes; these different categories were correlated with different YOB search ranges. However, a mechanism for making greater distinctions will provide greater flexibility in delimiting the set to be retrieved during the first stage of record analysis. The introduction of five refusal code levels also provides the opportunity to correlate more year-of-birth ranges to the refusal code levels.

3.8.4. Function

The RCL provides information needed for the evaluation of record proximity in the Arabic filtering process and contributes to the delimitation of database records retrieved through the RL/YOB Data Store.

3.9. YEAR-OF-BIRTH RANGE DATA STORE DECOMPOSITION

3.9.1. Identification

This data store is known as the Year-of-Birth Range Data Store (YR).

3.9.2. Type

3.9.2.1. It is recommended that the YR be a parameter file, which can be accessed by the client so YOB ranges can be set. Alternatively, it could be represented as a system parameter whose value(s) are set in an .ini file.

3.9.2.2. The YR will define the YOB ranges that will be associated with a Refusal Level (see Section 3.8).

3.9.2.3. This data store has the following format:

Figure 26: Format: Year-of-Birth Range Data Store (YR)

DATA FIELD	DATA TYPE	FIELD SIZE	VALUE	DATA DEFINITION
YOB0	integer	1	0	exact date of birth
YOB1	character	1	A	exact year, inverted month and day
YOB2	character	1	B	exact year of birth
YOB3	integer	2	1...99	narrow year of birth range
YOB4	integer	2	1...99	standard year of birth range
YOB5	integer	2	1...99	wide year of birth range
YOB6	integer	2	1...99	unlimited year of birth range

3.9.2.3.1. Definitions

3.9.2.3.2. DATA FIELD: YOB# is the Year-of-Birth Range category whose value indicates the year-of-birth range to be searched. The year-of-birth VALUE indicates the search range, that is, the number of years on either side of a given year-of-birth to be searched. For example, if the input year is 1962 and YOB3 range is 4, the search will cover a range of nine years, 1958-1966. The range includes the full year, so all of 1958 and all of 1966.

3.9.2.3.2.1. There are seven YOB# categories, YOB0, YOB1, YOB2, YOB3, YOB 4, YOB5, YOB6.

- **YOB0** is a single integer that refers to an exact month, day, year of birth. If YOB0 is specified, the system must be able to match the month, day and year of the Date of Birth of an input record and a database record.
- **YOB1** is a single character (A) that refers to an exact year-of-birth with the month and day inverted.
 1. If YOB1 is specified, the system must be able to match the year of Date of Birth and an inverted month and day (DEC 03 → MAR 12) of the input record and the database record.
 2. YOB1 will be relevant to the Arabic Filter and Sorter, but may not function as a search parameter since the value would be subsumed in YOB2.
- **YOB2** is a single character (B) that refers to an exact year-of-birth. If YOB2 is specified, the system must be able to match the year of the Date of Birth of an input record and a database record.
- **YOB3** is a one- or two-place integer (1...99) that refers to a narrow year-of-birth range. Narrow year-of-birth range is usually defined as 1 year (for a search range of 3 years).

- **YOB4** is one- or two-place integer (1...99) that refers to a standard year-of-birth range. Standard year-of-birth range is usually defined as 3 years (for a search range of 7 years).
- **YOB5** is a one- or two-place integer (1...99) that refers to a wide year-of-birth range. Wide year-of-birth range is usually defined as 5 years (for a search range of 11 years).
- **YOB6** is a one- or two-place integer (1...99) that refers to an unlimited or extremely wide year-of-birth range. Unlimited year-of-birth range would be set sufficiently high to include all (or all desired) years-of-birth in the database (e.g., 50).

3.9.3. Purpose

This YR provides a greater granularity in the year-of-birth range and, therefore, greater flexibility in delimiting the set to be retrieved during the first stage of record analysis. The correlation of five refusal code levels to different year-of-birth ranges will help to delimit the number of records to be searched and to define the more valuable set of records.

3.9.4. Function

- 3.9.4.1. The YR permits greater granularity in the Date-of-Birth types related to the system.
- 3.9.4.2. The YR will be accessed by the Refusal Code Level/YOB Range Data Store, which will limit the retrieval range in the Arabic Search Engine.
- 3.9.4.3. The YR will contribute to the Arabic Filter and Sorter to contribute information to the composite score.

3.10. REFUSAL CODE LEVEL / YOB RANGE DATA STORE MODULE DECOMPOSITION

3.10.1. Identification

This data store is known as the Refusal Code Level/YOB Range Data Store (RLYOB).

3.10.2. Type

- 3.10.2.1. The RLYOB is a matrix that merges the values in the Refusal Code Level (RCL) Data Store and the Year-of-Birth Range (YR) Data Store.
- 3.10.2.2. For each Refusal Level (RL), a Year-of-Birth (YOB) Range is specified.
 - 3.10.2.2.1. Only one YOB Range for each RL is permitted.
 - 3.10.2.2.2. The same YOB Range may apply to more than one RL.

3.10.2.3. The RLYOB has the following format:

Figure 27: Format: Refusal Level/Year-of-Birth Range Data Store (RLYOB)

DATA FIELD	DATA TYPE	FIELD SIZE	VALUE RANGE	DATA VALUE
RL#	character	3	RL0...4	RL0, RL1, RL2, RL3, RL4
YOB#	character	4	YOB0...6	YOB0, YOB1, YOB2, YOB3, YOB4, YOB5, YOB6

Figure 28: Example: RLYOB Data Store

RL#	YOB#
RL0	YOB5
RL1	YOB4
RL2	YOB3
RL3	YOB3
RL4	YOB2

3.10.2.4. Definitions:

3.10.2.5. RL#: is a character string that indicates the Refusal Level of the Refusal Code.

3.10.2.6. YOB#: is a character string that indicates the Date-of-Birth Range Category of the comparands.

3.10.3. Purpose

Retrieval of records from the database should be delimited by a relationship between the Refusal Code Level and the Year-of-Birth Range. It will restrict the number of records to be reviewed.

3.10.4. Function

The RLYOB is a resource for the Arabic Search Engine to delimit the records retrieved from the database.

3.11. COUNTRY-OF-BIRTH PROXIMITY DATA STORE DECOMPOSITION

3.11.1. Identification

This data store is known as the Country-of-Birth Proximity Data Store (COBPROX).

3.11.2. Type

3.11.2.1. The COBPROX is a matrix whose cells contain a decimal that reflects the degree of relationship between the country represented on the x-axis and the country represented on the y-axis.

3.11.2.2. The COBPROX has the following format:

Figure 29: Format: COBPROX Data Store

DATA FIELD	DATA TYPE	FIELD SIZE	VALUE RANGE	DATA VALUE
COB#1	character	4	alphabetic	COB Code
COB#2	character	4	alphabetic	COB Code
COBVAL	decimal	4	.00...1.00	Various

Figure 30: Example: Piece of COBPROX Data Store

COB#1	COB#2	COBVAL
AGS	AGS	1.00
AGS	ALG	0.05
AGS	MORO	0.05
AGS	SARB	0.05
AGS	SYR	0.05
ALG	ALG	1.00
ALG	MORO	0.85
ALG	SARB	0.75
ALG	SYR	0.75
MORO	MORO	1.00
MORO	SARB	0.75
MORO	SYR	0.75
SARB	SARB	1.00
SARB	SYR	0.75
SYR	SYR	1.00
...		

3.11.2.3. Definitions:

3.11.2.3.1. COB#1: is the 4-character COB Code of one of the comparands.

3.11.2.3.2. COB#2: is the 4-character COB Code of one of the comparands.

3.11.2.3.3. COBVAL: is the decimal value assigned through the ACOB (and other COB Category Data Stores).

3.11.3. Purpose

The COBPROX Data Store provides information on the relative value of the COBs in two comparands. This value can serve to limit the COBs that are accessed for retrieval.

3.11.4. Function

The COBPROX is populated by the ACOB and any other partition-specific Country-of-Birth Category Data Stores. The COBPROX provides COB relationship information.

3.12. ARABIC COUNTRY-OF-BIRTH CATEGORY DATA STORE DECOMPOSITION

3.12.1. Identification

This data store is known as the Arabic Country-of-Birth Category Data Store (ACOB).

3.12.2. Type

This ACOB is a data store that will serve as the source of information for the COBPROX Data Store, supplying the COBVAL, and will provide the COB Category (COBCAT) necessary for the Arabic Filter and Sorter.

Figure 31: Format: Arabic Country-of-Birth Category Data Store (ACOB)

DATA FIELD	DATA TYPE	FIELD SIZE	VALUE RANGE	DATA VALUE
COB#1	characters	4	alphabetics	COB Code
COB#2	characters	4	alphabetics	COB Code
COBCAT	characters	5	alphanumeric	COB1...COB99
COBVAL	decimal	4	0.00...1.00	Various

3.12.3. Definitions

3.12.3.1. COB#1: is the 4-character COB Code of one of the comparands.

3.12.3.2. COB#2: is the 4-character COB Code of one of the comparands.

3.12.3.3. COBCAT: is the category assigned to the relationship of two COBs.

3.12.3.3.1. Categories might include such relationships as Exact, State, Geographic Region, Dialect Region.

3.12.3.3.2. All relationships are adjustable.

3.12.4. COBVAL: is the decimal value that will be assigned to a particular COB relationship; this value will be used to determine the COBs that will be permitted in the retrieval process.

3.12.5. Example COB Categories might be:

COB1: Exact represents an exact match of the COBs: ALG/ALG; the COBPROXVAL would be 1.00.

COB2: Western Dialect Region represents the set of COBs that are in close geographic proximity and share naming conventions: ALG/MORO. The score would be something less than that applied to an exact match but nonetheless high: 0.85.

COB3: Arabic Partition represents all COBs within the Arabic partition. The value assigned would be less than that for COB2: 0.75.

COB4: All refers to all COBs and is assigned a value that will allow the search of all COBs; it would be the lowest decimal value used.

3.12. ARABIC COUNTRY-OF-BIRTH CATEGORY DATA STORE DECOMPOSITION

3.12.1. Identification

This data store is known as the Arabic Country-of-Birth Category Data Store (ACOB).

3.12.2. Type

This ACOB is a data store that will be serve as the source of information for the COBPROX Data Store, supplying the COBVAL, and will provide the COB Category (COBCAT) necessary for the Arabic Filter and Sorter.

Figure 31: Format: Arabic Country-of-Birth Category Data Store (ACOB)

DATA FIELD	DATA TYPE	FIELD SIZE	VALUE RANGE	DATA VALUE
COB#1	characters	4	alphabetics	COB Code
COB#1	characters	4	alphabetics	COB Code
COBCAT	characters	5	alphanumeric	COB1...COB99
COBVAL	decimal	4	0.00...1.00	Various

3.12.3. Definitions

3.12.3.1. COB#1: is the 4-character COB Code of one of the comparands.

3.12.3.2. COB#2: is the 4-character COB Code of one of the comparands.

3.12.3.3. COBCAT: is the category assigned to the relationship of two COBs.

3.12.3.3.1. Categories might include such relationships as Exact, State, Geographic Region, Dialect Region.

3.12.3.3.2. All relationships are adjustable.

3.12.4. COBVAL: is the decimal value that will be assigned to a particular COB relationship; this value will be used to determine the COBs that will be permitted in the retrieval process.

3.12.5. Example COB Categories might be:

COB1: Exact represents an exact match of the COBs: ALG/ALG; the COBPROXVAL would be 1.00.

COB2: Western Dialect Region represents the set of COBs that are in close geographic proximity and share naming conventions: ALG/MORO. The score would be something less than that applied to an exact match but nonetheless high: 0.85.

COB3: Arabic Partition represents all COBs within the Arabic partition. The value assigned would be less than that for COB2: 0.75.

COB4: All refers to all COBs and is assigned a value that will allow the search of all COBs; it would be the lowest decimal value used.

3.12. ARABIC COUNTRY-OF-BIRTH CATEGORY DATA STORE DECOMPOSITION

3.12.1. Identification

This data store is known as the Arabic Country-of-Birth Category Data Store (ACOB).

3.12.2. Type

This ACOB is a data store that will serve as the source of information for the COBPROX Data Store, supplying the COBVAL, and will provide the COB Category (COBCAT) necessary for the Arabic Filter and Sorter.

Figure 31: Format: Arabic Country-of-Birth Category Data Store (ACOB)

DATA FIELD	DATA TYPE	FIELD SIZE	VALUE RANGE	DATA VALUE
COB#1	characters	4	alphabetic	COB Code
COB#2	characters	4	alphabetic	COB Code
COBCAT	characters	5	alphanumeric	COB1...COB99
COBVAL	decimal	4	0.00...1.00	Various

3.12.3. Definitions

3.12.3.1. COB#1: is the 4-character COB Code of one of the comparands.

3.12.3.2. COB#2: is the 4-character COB Code of one of the comparands.

3.12.3.3. COBCAT: is the category assigned to the relationship of two COBs.

3.12.3.3.1. Categories might include such relationships as Exact, State, Geographic Region, Dialect Region.

3.12.3.3.2. All relationships are adjustable.

3.12.4. COBVAL: is the decimal value that will be assigned to a particular COB relationship; this value will be used to determine the COBs that will be permitted in the retrieval process.

3.12.5. Example COB Categories might be:

COB1: Exact represents an exact match of the COBs: ALG/ALG; the COBPROXVAL would be 1.00.

COB2: Western Dialect Region represents the set of COBs that are in close geographic proximity and share naming conventions: ALG/MORO. The score would be something less than that applied to an exact match but nonetheless high: 0.85.

COB3: Arabic Partition represents all COBs within the Arabic partition. The value assigned would be less than that for COB2: 0.75.

COB4: All refers to all COBs and is assigned a value that will allow the search of all COBs; it would be the lowest decimal value used.

Figure 32: Example: Piece of ACOB (Values for example only.)

COB#1	COB#2	COBCAT	COBVAL
ALG	ALG	COB1	1.00
ALG	MORO	COB2	0.85
ALG	SARB	COB3	0.75
ALG	SYR	COB3	0.75
MORO	MORO	COB1	1.00
MORO	SARB	COB3	0.75
MORO	SYR	COB3	0.75
SARB	SARB	COB1	1.00
SARB	SYR	COB3	0.75
SYR	SYR	COB1	1.00

3.12.6. Purpose

Pre-defined COB category relationships will provide a definition of the values that appear in the COBPROX Data Store.

3.12.7. Function

These COB categories will provide information about COB relationships that will contribute to determination of the Composite Score in the Arabic Filter and Sorter.

SOFTWARE DESIGN DESCRIPTION FOR THE HISPANIC NAME SEARCH ALGORITHM (HNA - E)

TABLE OF CONTENTS

1. INTRODUCTION	1
1.1. Purpose	1
1.2. Scope	2
1.3. Definitions and Acronyms	3
2. PROCESS FLOW	4
3. MODULE DECOMPOSITION	5
3.1. HISPANIC NAME SEARCH ALGORITHM FOR CLASS-E MODULE DECOMPOSITION	5
3.2. HISPANIC NAME PREPROCESSOR MODULE DECOMPOSITION	11
3.3. NAME LENGTH DETERMINER MODULE DECOMPOSITION	12
3.4. HISPANIC SURNAME SEGMENTER MODULE DECOMPOSITION	12
3.5. HISPANIC TITLE/AFFIX/QUALIFIER (TAQ) PROCESSOR MODULE DECOMPOSITION	14
3.6. HISPANIC SEGMENT POSITIONER MODULE DECOMPOSITION	17
3.7. HISPANIC NAME FORMATTER MODULE DECOMPOSITION	17
3.8. SEGMENT POSITION IDENTIFIER MODULE DECOMPOSITION	19
3.9. HISPANIC GENDER IDENTIFIER MODULE DECOMPOSITION	19
3.10. FREQUENCY PATH DIRECTOR MODULE DESCRIPTION	21
3.11. HIGH FREQUENCY PROCESSOR MODULE DECOMPOSITION	24
3.12. LOW FREQUENCY PROCESSOR MODULE DECOMPOSITION	31
3.13. HISPANIC SEARCH ENGINE MODULE DECOMPOSITION	45
3.14. HISPANIC FILTER AND SORTER MODULE DECOMPOSITION	51
3.15. LINGUISTIC TRACE FACILITY MODULE DECOMPOSITION	66
4. DATA DECOMPOSITION	67
4.1. DATA	67
4.2. DATA COLLECTION	68
4.3. DATA STORES	68
4.4. HISPANIC TITLE/AFFIX/QUALIFIER DATA STORE DECOMPOSITION	68
4.5. HIGH FREQUENCY SURNAME TYPE DATA STORE DECOMPOSITION	70
4.6. HIGH FREQUENCY SURNAME VARIANT DATA STORE DECOMPOSITION	72
4.7. HIGH FREQUENCY DECISION MATRIX DATA STORE	74
4.8. HISPANIC GIVEN NAME TYPE DATA STORE DECOMPOSITION	77

4.9. HIGH FREQUENCY GIVEN NAME VARIANT DATA STORE DECOMPOSITION	79
4.10. LOW FREQUENCY SURNAME TYPE DATA STORE DECOMPOSITION	80
4.11. HISPANIC CHARACTER DATA STORE.....	82
4.12. TAQ FILTER DATA STORE DECOMPOSITION	83
4.13. HISPANIC PARAMETER DATA STORE DECOMPOSITION	84
4.14. REFUSAL CODE CATEGORY DATA STORE DECOMPOSITION	86
4.15. YEAR-OF-BIRTH RANGE DATA STORE DECOMPOSITION	88
4.16. REFUSAL CODE LEVEL / YOB RANGE DATA STORE MODULE DECOMPOSITION	91
4.17. COUNTRY-OF-BIRTH PROXIMITY DATA STORE	92
4.18. HISPANIC COUNTRY-OF-BIRTH CATEGORY DATA STORE DECOMPOSTION.....	93

SOFTWARE DESIGN DESCRIPTION FOR THE HISPANIC NAME SEARCH ALGORITHM (HNA - E)

1. INTRODUCTION

1.1. Purpose

The current VLDB consists of about 5 million refusal records. The outlook envisions significant growth of the database in the near future and continued growth as more and more data are shared with other Government agencies. Currently, between 45% and 50% of the records have a country of birth from the Hispanic world, about 2.5 million records. These proportions are unlikely to change as the database expands.

Additionally, the character of Hispanic personal names is such that they are both *dense* and *complex*. *Dense* means that there are a relatively few individual surnames that account for the vast majority of surname occurrences. That is, the 500 most frequently occurring distinct surnames account for over 70% of all distinct surnames in the database. The surnames of well over 50% of the records contain only high frequency surnames. Another 25-30% contain at least one of the high frequency surnames. *Complex* means that Hispanic surnames generally contain more than 1 surname, the first of which is the family name, the second a matronymic (FLORES GOMEZ). Approximately 75% of the surnames from the Hispanic partition contain 2 surname stems (not including affixes like DE, DE LOS). Another 23% have only 1 surname stem. (The remaining records have 3-6 stems.)

The frequency of the names, the high portion of the VLDB and the syntactic variation that can occur in these names (inversion of the names, deletion of a name) argue for special handling of the Hispanic name search process.

The most important aspect of this specialized Hispanic name search algorithm is an efficient High Frequency Name Processor. Retrieval of fewer records for evaluation, yet ones that reflect some variation, is the goal of the High Frequency Processor.

The High Frequency Processor (HFP) of the HNA-E system targets the efficient processing of the most frequently occurring records in the Hispanic portion of the database. Early attempts at developing a processor that would handle high frequency Hispanic names had several major weaknesses.

- The earlier processors did not adequately address the characteristics of Hispanic names. In the name of performance, they did not allow for any variation in high frequency names.
- There was only one access method to the high frequency processor, which eliminated the processing of names similar to the high frequency names by the High Frequency Processor.
- Strict, often unmotivated, limitations were placed on the high frequency retrieval process. Little to no spelling or syntactic variation was permitted.
- The number of records retrieved was often extremely high, which resulted in a significant amount of post-processing.

All of these issues have been addressed in the HNA-E design. The HFP will be primarily list-based but the lists are empirically developed. It will identify and store relevant information about names, variants and their degree of proximity and will apply record similarity criteria *before* retrieval.

Low frequency Hispanic names, on the other hand, carry more information value because they are less usual. However, even low frequency names occur with sufficient frequency to challenge the system; the Hispanic database in general is very large. Preprocessing low frequency names will, therefore, also help reduce the number of records retrieved by limiting the search criteria.

1.2. Scope

The HNA-E system is intended to provide special and unique handling for names identified as Hispanic by the Automatic Name Classifier (ANC-E). It addresses the problem of highly frequent names to maximize retrieval potential and minimize the impact on performance and handles less frequently occurring names differently to accommodate the greater information content in these names. It also allows for broad variation in low frequency names and identifies potentially relevant records before database retrieval.

The input into the HNA-E system will be the output of the Advanced Name Classifier (ANC-E). ANC-E will determine if a name is Hispanic and therefore will undergo special processing by the Hispanic Name Algorithm (HNA-E). The design description of the ANC-E is contained as Attachment A in LAS Linguistic Memorandum CT970044 (May 30, 1997).

It became clear during the research for this design that the data stores that would be seminal to this system were very large. The Low Frequency Surname Type Data Store, for example, has over 90,000 records in it. Well over 37,000 of these names occur one time in the database; many of these are obvious misspellings or truncations of names. That is, the character strings do not occur in Spanish: RRODRIGUEZ, for example. It is suggested that a program of data stewardship be initiated to increase the efficiency of the system and reduce the storage needed for deviant material. One method of introducing data stewardship at this juncture would be to introduce Base

Records for the database records with errors and make the current database record of the new Base Record.

1.3. Definitions and Acronyms

ANC-E	Advanced Name Classifier for CLASS-E
DELETE	The name segment is completely disregarded in the remainder of the name search process and contributes minimal information to the record evaluation process; do not remove the segment from the record
DISREGARD	The name segment is disregarded in the remainder of the name search process but contributes to the evaluation of the name in the record evaluation process; do not remove the segment from the record
DI_KEY	Digraph Key (low frequency name types)
DI_VAL	Digraph Value (two-place decimal indicating digraph relation of two comparands.
F	Female
FNU	First Name Unknown
FPD	Frequency Path Director
FTI	Frequency Type Identifier
GN	Given Name
GNDR	Name Gender
GNTHR	Given Name Threshold (filter qualification)
GN_INIT	Given Name Initial Key
GN_VAL	Final Given Name Value
HCD	Hispanic Character Data Store
HDM	Hispanic Decision Matrix
HFGN_KEY	High Frequency Given Name Key (SET_ID of the GN_TYPE)
HFGV	High Frequency Given Name Variant Data Store
HFGN_VAR	High Frequency Given Name Variant Key
HF	High Frequency
HFP	High Frequency Processor
HFS	Hispanic Filter and Sorter
HFSN_KEY	High Frequency Surname Key (SET_ID of the HFSN_TYPE)
HFSN_VAR	High Frequency Surname Variant Key (ID_NO of the HFSN_VAR)
HFST	High Frequency Surname Type Data Store
HFSV	High Frequency Surname Variant Data Store
HGI	Hispanic Gender Identifier
HGT	Hispanic Given Name Type Data Store
HNA-E	Hispanic Name Search Algorithm for CLASS-E
HNF	Hispanic Name Formatter
HNP	Hispanic Name Preprocessor
HNT	Hispanic Given Name Type Data Store
HPD	Hispanic Parameter Data Store
HR	Hispanic Regularization Rule Base
HRE	Hispanic Rule Engine
HSE	Hispanic Search Engine
HSP	Hispanic Segment Positioner
HSS	Hispanic Surname Segmenter
HTD	Hispanic TAQ Data Store
HTP	Hispanic TAQ Processor
ID_NO	Identification Number for Segments in Data Stores
INITGN	Given Name Initial Parameter Value

INITNM	No Match Initial Parameter Value
INITSN	Surname Initial Parameter Value
LFDIKEY	Low Frequency Digraph Key in LFST
LFGT	Low Frequency Given Name Type Data Store
LFP	Low Frequency Processor
LFST	Low Frequency Surname Type Data Store
LF_DI THRESHOLD	Low Frequency Digraph Threshold
LNU	Last Name Unknown
LTF	Linguistic Trace Facility
M	Male
NLD	Name Length Determiner
REMOVE	A segment that is conjoined to the name stem is removed from the stem; it will then be marked for additional handling. DELETE or DISREGARD.
RGNDR	Record Gender
RL#	Refusal Code Level Category Number
SET_ID	Identification Number for Related Set of Name Variants
SEGMENT	Name element surrounded by white space
SN	Surname
SNTHR	Surname Threshold (filter qualification)
SN_INIT	Surname Initial Key
SN_VAL	Final Surname Value
SPI	Segment Position Identifier
TAQ	Title/Affix/Qualifier
TAQDIS#1	TAQ DISREGARD Comparand #1
TAQDIS#2	TAQ DISREGARD Comparand #2
U	Unknown/Ambiguous Name Gender
YOB	Year-of-Birth
YOB#	Year-of-Birth Range Category Number
YR	Year-of-Birth Range Data Store

2. PROCESS FLOW

A Hispanic name is pre-processed and prepared for key generation. Prefixes are removed, certain name segments are moved, record gender is determined and other name characteristics are collected.

The processor to which a name is submitted is dependent on the frequency of the surname, high frequency or low frequency. There are multiple entries into the High Frequency Processor, which means that low frequency names that are related to high frequency names can also be treated as high frequency names.

The underlying principle behind the handling of high frequency names is that they retrieve a specified set of variants, all of which have pre-determined digraph values associated with them. This places the processing burden on adding records to the system and reduces the burden at the time of the query. Record retrieval criteria have been defined according to the values of the names and their relative positions in the query string; a query with high frequency names will, therefore, retrieve a smaller set of relevant names. The goal is to retrieve an adequate range of names as rapidly as possible.

Variants of low frequency names will be identified before retrieval based on matching digraph keys. The system will then retrieve exact matches on the set of low frequency names that pass a low frequency threshold.

3. MODULE DECOMPOSITION

3.1. HISPANIC NAME SEARCH ALGORITHM FOR CLASS-E MODULE DECOMPOSITION

3.1.1. Identification

This program is known as the Hispanic Name Search Algorithm for CLASS-E (HNA-E)

3.1.2. Type

This program is a subprogram of the CLASS-E system and will process Hispanic names for both queries and record adds.

3.1.3. Purpose

HNA-E will process input names identified as Hispanic by the ANC-E using techniques that are appropriate for Hispanic names. No names with Last Name Unknown (LNU) will be processed by HNA-E.

3.1.4. Function

The Hispanic Name Search Algorithm for CLASS-E (HNA-E) consists of three program modules:

- the Hispanic Name Preprocessor (HNP),
- the Hispanic Search Engine (HSE), and
- the Hispanic Filter and Sorter (HFS).

3.1.4.1. The HNP will manipulate an input name to generate search keys, generate additional query forms or alias record adds, calculate record gender, collect information about the input name and its name segments and determine the frequency path to which a name will be submitted for processing.

3.1.4.1.1. The HNP will pass an input name to one of two processing paths:

- the High Frequency Name Processor (HFP) or
- the Low Frequency Name Processor (LFP).

3.1.4.1.2. The HNP will generate a set of record criteria and search keys for retrieval of records from the database.

3.1.4.2. The HSE will build the retrieval keys, extract record information relevant to the retrieval and retrieve database records according to the keys and criteria identified.

3.1.4.3. The HFS will evaluate the database records and will prepare an ordered set of records for return to the user.

3.1.4.3.1. The HFS will qualify records based on filtering criteria and parameters.

3.1.4.3.2. The HFS will sort the qualifying database records into an ordered list with the names most closely proximate to the query name at the top.

3.1.5. Subordinates

HNA-E consists of 3 major programming modules: (See Pages 7-10 for graphic representations of the processing flow of these modules.)

- Hispanic Name Preprocessor (HNP),
- Hispanic Search Engine (HSE), and
- Hispanic Filter and Sorter (HFS).

3.2. HISPANIC NAME PREPROCESSOR MODULE DECOMPOSITION

3.2.1. Identification

This module is known as the Hispanic Name Preprocessor (HNP).

3.2.2. Type

The HNP is a subprogram of the HNA-E program that accepts input from the Advanced Name Classifier for CLASS-E (ANC-E) and prepares it for handling by the Hispanic Search Engine (HSE). (See Section 3.13.)

3.2.3. Purpose

Hispanic names account for almost 50% of the VLDB name records. In addition to the volume of occurrence, there are many names that occur very frequently. The format of Hispanic names contributes further obstacles to name searching: the surname generally consists of two names and the given names generally consists of two names. The most highly frequently occurring prefix in the VLDB is also Hispanic: DE. The frequency, density and the nature of the name argue for preparing the name in whatever way(s) are necessary to expedite the retrieval process. That is the function of the HNP.

3.2.4. Function

3.2.4.1. The HNP will prepare a name identified as Hispanic by the ANC-E for the HSE by

- identifying name segments and determining their disposition,
- manipulating the name segments to generate additional query formats,
- determining name length and record gender,
- specifying the frequency character of each name segment and
- generating search keys.

3.2.4.2. Because of the significant amount of information that is to be generated and collected about the name through the HNP, it is strongly recommended that the name be treated as an object that "knows" what sorts of information it needs. Such an object will provide a mechanism for following the acquisition of information as the object passes through the system. Much of that information will be collected and loaded during the HNP stage.

3.2.5. Subordinates

The HNP has ten subordinate functions:

- Name Length Determiner (NLD)
- Hispanic Surname Segmenter (HSS)
- Hispanic TAQ Processor (HTP)
- Hispanic Segment Positioner (HSP)
- Segment Position Identifier (SPI)
- Hispanic Name Formatter (HNF)
- Hispanic Gender Identifier (HGI)

- Frequency Path Director (FPD)
- High Frequency Name Processor (HFP)
- Low Frequency Name Processor (LFP)

3.3. NAME LENGTH DETERMINER MODULE DECOMPOSITION

3.3.1. Identification

This function is known as the Name Length Determiner (NLD).

3.3.2. Type

The NLD is a function that accepts as input a surname (SN) segment and stores the surname length. The length will be used by the Hispanic Surname Segmenter (Section 3.4).

3.3.3. Purpose

Name segment length will provide information that will be used by the Hispanic Surname Segmenter to attempt to divide surnames over a specific length into component segments.

3.3.4. Function

3.3.4.1. The NLD will accept as input each SN segment.

3.3.4.1.1. A segment is a string of characters surrounded by white space.

3.3.4.1.2. The NLD will count the number of characters in SN segment (not including surrounding blanks).

3.3.4.1.3. The NLD will store the length count associated with each SN segment.

3.3.5. Subordinates

None.

3.4. HISPANIC SURNAME SEGMENTER MODULE DECOMPOSITION

3.4.1. Identification

This function is known as the Hispanic Surname Segmenter (HSS).

3.4.2. Type

The HSS attempts to divide surnames over a specified length into component segments. The HSS is a function that must follow the NLD and precede the Hispanic TAQ Processor (Section 3.5).

3.4.3. Purpose

Hispanic names often have many segments and these segments may be quite long. Field lengths of fixed size may not be able to accommodate the number of name segments that occur. Data entry operators often attempt to reduce the name length by conjoining name segments. Conjoined segments have an especially negative impact on the surname. The access point into the database

is through the surname and conjoined name segments generally make the component segments inaccessible to processing. Separating conjoined surnames would, therefore, improve the search process.

3.4.4. Function

3.4.4.1. The HSS will separate conjoined HF SN segments from a surname segment of nine characters or more in length.

3.4.4.1.1. The HSS will generate additional query records for the separated SN segment and tag the items separated.

3.4.4.1.2. The HSS will generate alias record adds for the separated SN name segments.

3.4.4.2. The HSS will access the High Frequency Surname Type Data Store (HFST).

3.4.4.2.1. Phase 1: The HSS will begin with the leftmost character of the query/add SN segment and attempt to identify a HFSN_TYPE within the input SN string.

3.4.4.2.2. The HSS will choose the longest HFSN_TYPE that it can identify, separate that string from the input string and proceed to Phase 2.

3.4.4.2.3. Phase 2: The HSS will begin with the rightmost character of the query/add SN segment and attempt to identify a HFSN_TYPE (in reverse order) within the remaining input string (after any HFSN_TYPE has been removed during Phase 1).

3.4.4.2.4. The HSS will choose the longest HFSN_TYPE that it can identify and separate that string from the remaining input string.

3.4.4.2.5. Any residual segment will be retained as is.

3.4.4.2.6. If no HFSN_TYPE can be identified in either Phase, no action will be taken.

3.4.4.2.7. An alias (or additional query) will be generated for the divided string.

Figure 1: Example: Hispanic Surname Segmenter

INPUT NAME	HFSN_TYPE	PHASE 1	PHASE 2	OUTPUT
GARCIAGOMEZ	GARCIA GOMEZ	GARCIA	GOMEZ	GARCIA GOMEZ
PEREZDELOPEZ	PEREZ LOPEZ	PEREZ	DELOPEZ	PEREZ DE LOPEZ
BOMEZDEPEREZ	PEREZ	BOMEZDE	PEREZ	BOMEZDE PEREZ
RAMIREZDELAPAZ	RAMIREZ PAZ	RAMIREZ	DELAPAZ	RAMIREZ DELA PAZ

3.4.5. Subordinates

None.

3.5. HISPANIC TITLE/AFFIX/QUALIFIER (TAQ) PROCESSOR MODULE DECOMPOSITION

3.5.1. Identification

This module will be known as the Hispanic Title/Affix/Qualifier Processor (HTP).

3.5.2. Type

The HTP is a process that accepts a full Surname (SN) or Given Name (GN), accesses the Hispanic TAQ Data Store and reduces name fields with multiple segments to their name stems.

3.5.3. Purpose

Hispanic names frequently contain peripheral name elements, such as DE, DE LA, DEL, SAN. Matching on these segments is not generally useful; the name segments with information value are the name stems. For example, GARCIA is the more valuable segment in the string DE GARCIA, as is ANGELES in DE LOS ANGELES. Removal of or disregard for the peripheral name elements allows more emphasis to be placed on the name stems, thus improving the search process.

3.5.4. Function

3.5.4.1. The HTP will access the Hispanic TAQ Data Store (HTD) to identify TAQ segments: titles (e.g., SR., MR.), affixes (e.g., DE) or qualifiers (e.g., PH.D., HIJO).

3.5.4.1.1. The HTD will contain information about the disposition of the TAQ.

3.5.4.1.2. The HTD will contain information about the type of TAQ (TAQ_TYPE): Title, Prefix, Infix, Suffix, Qualifier.

3.5.4.2. The HTP will scan all SN segments or all GN segments for any TAQ segments.

3.5.4.2.1. The HTP will begin with the leftmost character of the SN or GN field and attempt to identify a TAQ segment among the SN segments and among the GN segments. (The TAQ segment will be surrounded by white space.)

3.5.4.2.2. If the HTP identifies a segment, it will tag the segment with the ID_NO and disposition, as indicated in the HTD.

3.5.4.2.3. If the following segment is also a TAQ segment, it will tag the segment with the ID_NO and disposition, as indicated in the HTD.

- 3.5.4.2.4. This will continue until all *consecutive* TAQ segments have been tagged.
- 3.5.4.2.5. When the HTP encounters a following segment that is not a TAQ segment, it will treat that segment as a stem.
 - 3.5.4.2.5.1. Each TAQ segment identified up to that point will be given the TAQ_TYPE P (prefix) and each will be associated and stored with the following stem.
- 3.5.4.2.6. The HTP will move to the next segment following the stem and will repeat the TAQ identification process.
 - 3.5.4.2.6.1. The HTP will tag all TAQ segments with the ID_NO and disposition.
 - 3.5.4.2.6.2. When the HTP encounters a stem, it will tag each TAQ segment (not yet associated with a stem) with the TAQ_TYPE P and will associate and store each TAQ segment with the following stem.
- 3.5.4.2.7. If HTP encounters a TAQ segment or segments that has no following stem, it will access the HTD to determine if the TAQ type is a Suffix (S).
 - 3.5.4.2.7.1. If the TAQ has a TAQ_TYPE S, the TAQ will be associated and stored with the *preceding* stem.
 - 3.5.4.2.7.2. The preceding stem may already have prefixal TAQs.
 - 3.5.4.2.7.3. If the TAQ type is not equal to S, the TAQ will be tagged a Stranded Prefix.
- 3.5.4.3. The HTP will process any TAQ segments identified according to the treatment indicated in the HTD.
- 3.5.4.4. Treatment options include DELETE, DISREGARD and REMOVE.
 - 3.5.4.4.1. **DELETE** means that the segment is completely disregarded in the remainder of the name search process and contributes marginal information to the filtering process. (N.B. The segment is not deleted from the record.)
 - 3.5.4.4.2. **DISREGARD** means that the segment is disregarded in the remainder of the name search process but contributes to the evaluation of the name in the filtering processes.
 - 3.5.4.4.3. **REMOVE** means that a segment that is conjoined to the name stem is removed from that stem. It is then submitted to additional handling, either DELETE or DISREGARD.
 - 3.5.4.4.3.1. The HTP will begin with the leftmost character in the input stem segment (after free-standing TAQs have

been removed) and will attempt to identify all TAQ segments that have been marked for removal (REMOVE).

3.5.4.4.3.2. The HTP will begin with the longest TAQ segment and attempt to remove that; it will then proceed to shorter segments.

3.5.4.4.3.3. If the segment that is to remain after TAQ removal is two characters or fewer, the HTP will not remove the TAQ.

3.5.4.4.3.4. If the TAQ segment is identified and the residual stem is of sufficient length, it is separated from the stem.

3.5.4.4.3.5. The HTP assigns and stores the ID_NO of the removed TAQ.

3.5.4.4.3.6. The HTP then submits the removed TAQ to the treatment indicated (DELETE or DISREGARD) in the HTD and tags and stores the TAQ with that treatment indicator.

Figure 2: Example: TAQ REMOVE Process

INPUT STRING	TAQ REMOVE	OUTPUT
DECORDOBA	DE	DE CORDOBA
DELOSANGELES	DELOS	DELOS ANGELES
DEARING	DE	DE ARING
MARIADE	DE	MARIADE
DELPILAR	DEL	DEL PILAR

3.5.4.5. After all TAQ segments have been submitted to the appropriate process, the remaining segments will be considered SN and GN stems.

Figure 3: Example: TAQ Processing

INPUT SN:	TAQs and STEMS	OUTPUT SN:
DE	TAQ: DE (ID_NO, REMOVE, DISREGARD)	
LA	TAQ: LA (ID_NO, REMOVE, DISREGARD)	
CRUZ	STEM: CRUZ	CRUZ
DE	TAQ: DE (ID_NO, REMOVE, DISREGARD)	
BARRIOS	STEM: BARRIOS	BARRIOS
SAN	TAQ: SAN (ID_NO, DISREGARD, Stranded Prefix)	

3.5.5. Subordinates

None.

3.6. HISPANIC SEGMENT POSITIONER MODULE DECOMPOSITION

3.6.1. Identification

This function is known as the Hispanic Segment Positioner (HSP).

3.6.2. Type

The HSP is a function that moves a high frequency (HF) surname (SN) found in the given name (GN) field into the SN field.

3.6.3. Purpose

Surnames that occur in the GN field deprive the match process of relevant SN information. Moving a SN segment that occurs in the GN field to the SN field will benefit the search process. (The SN segment is moved to the rightmost position to retain the value assigned to the resident SN segment(s).)

3.6.4. Function

3.6.4.1. If more than one GN segment (stem) occurs in the GN field, the HSP will determine if the final (rightmost) segment in the GN string is a HF SN.

3.6.4.2. The HSP will move the segment to the SN field.

3.6.4.2.1. If more than one GN segment occurs in the GN field, the HSP will access the High Frequency Surname Type Data Store (HFST) to determine if the rightmost GN segment is a HFSN_TYPE.

3.6.4.2.2. If the segment is a HFSN_TYPE, the HSP will move the segment into the rightmost position of the SN field.

3.6.4.3. The process applies to one name segment only and is not iterative.

Figure 4: Example: Hispanic Segment Positioner (HSP)

INPUT NAME	HFSN_TYPE	OUTPUT FORMAT
CASTRO, MARIA LUZ GOMEZ	GOMEZ	CASTRO GOMEZ, MARIA LUZ
BARRIOS LUNA, JUAN PEREZ	PEREZ	BARRIOS LUNA PEREZ, JUAN
LOPES ARRIAGA, CARLOS VITRAL	LOPES ARRIAGA, CARLOS VITRAL

3.6.4.4. An additional query record is generated with the moved segment; the original record is not changed.

3.6.4.5. An alias record add is generated with the moved segment; the original record is not changed.

3.6.5. Subordinates

None.

3.7. HISPANIC NAME FORMATTER MODULE DECOMPOSITION

3.7.1. Identification

This module is known as the Hispanic Name Formatter (HNF).

3.7.2. Type

3.7.2.1. The HNF is a process that generates additional name formats for input records that have more than two surname stems.

3.7.2.2. The HNF will follow the HSS, HTP, and HSP.

3.7.2.3. The generated formats will serve as the name format for HF name processing and for comparison in the filtering and sorting process.

3.7.3. Purpose

The HNF will limit the number of segments that can occur in the surname field to two in order to maximize the efficient processing of the input name.

3.7.4. Function

3.7.5. The HNF will generate additional alias record adds and queries for surnames that contain more than two SN stems.

3.7.5.1. The HNF will accept input strings with any number of SN segments (stems).

3.7.5.2. When more than two SN segments are present, the HNF will generate additional name formats with a limit of two SN segments.

3.7.5.2.1. The HNF will begin with the leftmost SN segment and generate dual-SN formats with each additional SN segment.

3.7.5.2.2. The HNF will move to the second SN segment and generate dual-SN formats with each other SN segment that have not yet been generated.

3.7.5.2.3. The relative order of all segments will be maintained.

3.7.5.3. All generated formats will be stored with the record add.

3.7.5.4. All generated formats will be additional queries.

Figure 5: Example: Hispanic Name Formatter (HNF)

INPUT SURNAME	GARCIA	LUNA	BUSTOS	ARRIAGA
HNF DUAL-SN FORMATS	GARCIA	LUNA		
	GARCIA		BUSTOS	
	GARCIA			ARRIAGA
		LUNA	BUSTOS	
		LUNA		ARRIAGA
			BUSTOS	ARRIAGA

3.7.6. Subordinates

None.

3.8. SEGMENT POSITION IDENTIFIER MODULE DECOMPOSITION

3.8.1. Identification

This module is known as the Segment Position Identifier (SPI).

3.8.2. Type

The SPI is a function that identifies the relative position of each of the SN and GN stems. The SPI must follow the HTP, HSP and HNF. Segment position information will be accessed by the High Frequency Processor (HFP) and the Hispanic Filter and Sorter (HFS).

3.8.3. Purpose

Hispanic names generally contain more than one SN and more than one GN. The value of each of these name stems is different. In a SN, the leftmost stem is the family name; other SN stems are differentiators. The family name carries more value in the SN. In a GN, the leftmost name stem generally indicates gender so is a valuable indicator. Names that are in- and out-of position are therefore of differing relevance. Position information can contribute to the selection and evaluation of relevant records.

3.8.4. Function

3.8.4.1. The SPI will operate on any SN or GN except where dual-SN formats have been generated.

3.8.4.1.1. Where dual-SN formats have been created, the SPI will accept only those formats.

3.8.4.1.2. The SPI will accept any number of GN segments.

3.8.4.2. The SPI will specify the position in the name field (SN or GN fields) of each name segment.

3.8.4.3. The SPI will begin with the leftmost segment and assign Position 1, proceeding to the next segment and assign Position 2, and so forth.

3.8.4.4. Position information will be generated for and stored with each SN segment.

3.8.4.5. Position information will be generated for and stored with each GN segment.

3.8.5. Subordinates

None.

3.9. HISPANIC GENDER IDENTIFIER MODULE DECOMPOSITION

3.9.1. Identification

This function is known as the Hispanic Gender Identifier (HGI).

3.9.2. Type

This is a function that associates a gender value with a *record*; it will be accessed by the Hispanic Decision Matrix (HDM) and the Hispanic Filter and Sorter (HFS).

3.9.3. Purpose

It is usually possible to predict the gender of a Hispanic name based on the gender marker of the leftmost given name segment. Because crossed-gender names are of little value in the visa adjudication process, lowering the value of a record whose gender does not match that of a query would improve the name matching process.

Predicting gender based on one source of gender, however, may result in elimination of records that differ by one character only. More than one source of gender information can provide a means of validating the gender assignment. This will be the record gender. Record gender will reduce the chance of qualifying or disqualifying a record based on the gender of a single name segment, which could be misspelled or ambiguous with respect to gender.

3.9.4. Function

3.9.4.1. The HGI will derive a gender that will be associated with a *record* and not a Given Name stem alone.

3.9.4.2. A record gender value may be Male (M), Female (F), or Unknown/Ambiguous (U) Gender.

3.9.4.2.1. The HGI will derive the record gender from the GN gender associated with each GN segment and the gender provided by the user during the data entry process.

3.9.4.2.1.1. The HGI will access the Hispanic Given Name Type Data Store (HGT) to determine the gender associated with each GN segment.

3.9.4.2.1.1.1. If the name is present in the HGT, the gender indicated will be associated with the GN segment.

3.9.4.2.1.1.2. If the name is not present in the HGT, the record gender will be marked as Unknown (U). (This would occur for a query with a name never before submitted to the system.)

3.9.4.2.1.2. The applicant gender is determined at the time of application and must be entered, captured and stored by the system.

3.9.4.2.2. The HGI will verify that all gender indicators agree: the gender associated with each GN segment and the applicant gender received at the time of application.

3.9.4.2.2.1. To mark the record gender as M or F, the HGI requires gender validation from a *minimum* of two sources.

3.9.4.2.2.2. All sources of gender information (whether two or more) must match for gender to be marked as M or F.

3.9.4.2.2.2.1. If the gender indicators match, the match value will become the record gender.

3.9.4.2.2.2.2. If the gender indicators do not match, gender is marked as U.

Figure 6: Example: Record Gender Assignment

GIVEN NAME	HGT GNDR	INPUT GENDER	RECORD GENDER
1) MARIA	F	F	F
LUZ	F		
2) JOSE	M	M	M
ANTONIO	M		
3) CARLOS	M	M	U
(DE LA) CRUZ	U		
4) BERNARDO	M	M	M
5) CAMEN (misspelling)	(not in HGT)	F	U
MARIA	F		

3.9.5. Subordinates

None.

3.10. FREQUENCY PATH DIRECTOR MODULE DESCRIPTION

3.10.1. Identification

This module is known as the Frequency Path Director (FPD).

3.10.2. Type

3.10.2.1. The FPD directs a record to the High Frequency Processor or Low Frequency Processor depending on the presence or absence of HF *surnames* in the string.

3.10.2.2. The FPD will access the following data stores:

- High Frequency Surname Type Data Store (HFST)
- Hispanic Character Data Store (HCD)

3.10.3. Purpose

Many Hispanic names occur with such high frequency that they would benefit from special processing. The system must determine which the high

frequency surnames are and direct records with high frequency surnames to the proper handler.

3.10.4. Function

3.10.4.1. The FPD will accept any SN format, except where dual-SN formats have been generated by the Hispanic Name Formatter (HNF).

3.10.4.1.1. The FPD will operate on the dual-SN formats where they have been generated.

3.10.4.2. The FPD will identify, process and assign keys to SN initials.

3.10.4.3. The FPD will identify and tag each SN stem as HF or LF.

3.10.4.4. The FPD will assign HFSN_KEYS, where appropriate.

3.10.4.5. The FPD will direct the record to the HF Processor or LF Processor depending on the frequency tags of the SN segments.

3.10.4.6. The frequency-identification process will repeat until the frequency value of all SN segments has been identified.

3.10.4.7. Surname Initials

3.10.4.8. Record Adds

3.10.4.9. The FPD will generate a SN_INIT Key for the initial character of each SN segment (The SN segment may be an initial).

3.10.4.9.1. The FPD will access the Hispanic Character Data Store (HCD) to identify the SN_INIT Key.

3.10.4.9.2. The FPD will find the initial character in the CHAR list.

3.10.4.9.3. The FPD will assign the SN_INIT Key to the character.

3.10.4.9.4. The SN_INIT Key will be the SET_ID for all occurrences of the character.

3.10.4.9.5. The FPD will store the SN_INIT Key with the SN segment of the record.

3.10.4.10. Query

3.10.4.11. The FPD will identify single characters that occur in the SN field; any segment that has a name length of 1 (as specified by the Name Length Determiner (NLD)) is an initial.

3.10.4.12. The FPD will access the Hispanic Character Data Store (HCD) to determine the SN_INIT Key(s) to assign to the segment.

3.10.4.12.1. The FPD will find each instance of the character in the CHAR_VAR list.

3.10.4.12.2. The FPD will assign SN_INIT Key(s) to the SN initial.

- 3.10.4.12.3. The SN_INIT KEY is the SET_ID associated with each instance of the initial.
- 3.10.4.12.4. The SN initial may have multiple SN_INIT Keys.
- 3.10.4.13. The FPD will ignore the SN_INIT Keys when determining the frequency path assignment of a record; the assignment will be based on the frequency of the other SN segment.
- 3.10.4.14. **High Frequency Surnames**
- 3.10.4.15. The FPD will access the High Frequency Surname Type Data Store (HFST).
- 3.10.4.16. If a SN segment matches exactly a HFSN_TYPE in the HFST, the segment will be given the HFSN_KEY associated with the HFSN_TYPE.
- 3.10.4.16.1. **Record Add/Query:** The HFSN_KEY will be the SET_ID associated with the HFSN_TYPE in the HFST.
- 3.10.4.16.2. **Record Add:** A digraph value (DI_VAL) of 1.00 will be assigned to and stored with the segment that matches a HFSN_TYPE exactly.
- 3.10.4.16.3. The HFSN_KEY will represent a set of name segments that have qualified as digraph variants of the HFSN_TYPE. (See 3.12.4.38 for information on how the variants are assigned to the same set.)
- 3.10.4.17. The FPD will direct records that contain *all* HFSN_KEYs to the High Frequency Processor (HFP).

Figure 7: Example: Qualification for High Frequency Processor

INPUT NAME:	FIELD	HFSN_KEY	DI_VAL
GARCIA LOPEZ, ANTONIO JESUS			
GARCIA	SN	0001	1.00
LOPEZ	SN	0004	1.00

Figure 8: Resource: Piece of High Frequency Surname Type Data Store (HFST)

ID_NO	HFSN_TYPE	SET_ID
0001	GARCIA	0001
0002	RODRIGUEZ	0002
0003	HERNANDEZ	0003
0004	LOPEZ	0004
0005	MARTINEZ	0005
0006	GONZALEZ	0006
0007	PEREZ	0007
0008	SANCHEZ	0008
0009	RAMIREZ	0009
0010	GOMEZ	0010
0011	...	0011

3.10.4.18. The FPD will direct all records that do *not* contain all HFSN_KEYS to the Low Frequency Processor (LFP).

3.10.5. Subordinates

None.

3.11. HIGH FREQUENCY PROCESSOR MODULE DECOMPOSITION

3.11.1. Identification

This module is known as the High Frequency Processor (HFP).

3.11.2. Type

3.11.2.1. The HFP is a program module that

- will process records with all HFSN_KEYS, HFSN_VAR Keys, SN_INIT Keys and mixed HF and LF Keys;
- will generate Given Name Keys; and
- will access the Hispanic Decision Matrix Data Store (HDM) to identify retrieval criteria for the HF records.

3.11.2.2. Multiple entry points into the HFP will be supported: through the Frequency Path Director and the Low Frequency Processor (LFP).

3.11.3. Purpose

Earlier attempts to develop a HF handler for Hispanic names have been limited to processing of records that contain only HF names; little to no variation was permitted. HNA-E will support variation in the processing of HF names by allowing multiple entry points into the HFP.

3.11.4. Function

3.11.4.1. The HFP will accept names directed to the processor by the Frequency Path Director (FPD) and by the LFP.

3.11.4.2. The records accepted will contain all HFSN_KEYs and/or HFSN_VAR Keys, SN_INIT Keys, and mixed HFSN_KEY/HFSN_VAR and DI_KEYs.

3.11.4.3. All SN_INIT Keys passed to the HFP will be treated as segment Keys and will undergo the same criteria identification as other segments.

3.11.4.4. If all segments of the SN Field have been given HFSN_KEYs and/or HFSN_VAR Keys and related DI_VALs, the HFP will begin processing the GN segments.

3.11.4.5. Processing the Given Name Segments

3.11.4.6. If the GN segment is First Name Unknown (FNU), no GN processing will take place.

3.11.4.7. High Frequency Given Name Segment Keys

3.11.4.8. Record Adds

3.11.4.9. The HFP will access the Hispanic Given Name Variant Data Store (HGNV) to determine if the GN segments are HF GN segments.

3.11.4.9.1. If the GN segment matches one or more variants in the HGNV, the HFP will assign the HFGN_KEY to the GN segment.

3.11.4.9.2. The HFGN_KEY is(are) the SET_ID(s) associated with the variant.

3.11.4.9.3. The HFP will associate the appropriate DI_VAL with the SET_ID and GN segment.

3.11.4.9.4. The HFP will store the SET_ID(s) and their DI_VAL with the GN segment.

3.11.4.9.5. The HFGN_KEY ensures that the system will retrieve variants of a HF segment when the HF segment is queried.

3.11.4.10. Query

3.11.4.11. The HFP will access the Hispanic Given Name Type Data Store (HGT).

3.11.4.11.1. If a GN segment matches exactly a GN_TYPE name segment in the HGT and HI_FREQ = 1 (is True) (that is, the segment is a HF GN_TYPE segment), the HFP will assign to the GN segment the HFGN_KEY associated with the GN_TYPE.

- 3.11.4.11.2. The HFGN_KEY will be the SET_ID associated with the HF GN_TYPE.
- 3.11.4.12. **High Frequency Given Name Initial Keys**
- 3.11.4.13. The HFP will create one or more GN_INIT Keys for each HF GN segment.
- 3.11.4.14. The GN_INIT Key will be the initial key for each GN segment, including initials.
 - 3.11.4.14.1. **Record Add**
 - 3.11.4.14.2. The HFP will identify the initial character of each GN segment.
 - 3.11.4.14.3. The HFP will access the Hispanic Character Data Store (HCD) and will find all occurrences of the character in the CHAR-VAR list.
 - 3.11.4.14.4. The HFP will assign the GN_INIT Key(s) to each GN initial.
 - 3.11.4.14.4.1. The GN_INIT Key will be the SET_ID(s) associated with the GN initial (CHAR_VAR).
 - 3.11.4.14.4.2. The GN segment may have multiple GN_INIT Keys.
 - 3.11.4.14.4.3. The GN_INIT Key will permit retrieval of multiple initials for a GN initial.
 - 3.11.4.14.4.4. The HFP will store the GN_INIT Key(s) for each GN segment initial with the record.
 - 3.11.4.14.5. Query
 - 3.11.4.14.6. The HFP will access the HCD and find the initial in the CHAR list.
 - 3.11.4.14.7. The HFP will identify the GN_INIT Key for each GN segment initial.
- 3.11.4.15. If the GN segment is not a variant in the HGNV, the HFP will tag the name as LF.
- 3.11.4.16. **Low Frequency Given Name Segment Keys**
- 3.11.4.17. For each LF GN segment, the HFP will attempt to determine if the segment is a potential variant of a HF GN_TYPE and will create one or more GN_INIT Keys for record adds and queries.
- 3.11.4.18. **Record Add**
- 3.11.4.19. If the LF GN is *not* in the HGNV Data Store, the HFP will determine if the segment is a potential variant of a HFGN_TYPE. (This

would apply to LF GN segments that are being submitted to the system for the first time.)

3.11.4.19.1. If the HFP determines that the HF GN segment is a variant of a HFGN_TYPE, the LFP will append the segment to the HFGV Data Store.

3.11.4.19.2. The HFP will access the Hispanic Given Name Type Data Store (HGT) to determine if the LF GN segment is a digraph variant of one or more of the HF GN_TYPES. (That is, the LF GN segment is a digraph variant of the GN_TYPE whose HF Value is True (1)).

3.11.4.19.3. The LFP will perform a digraph evaluation of the LF GN and each HF GN_TYPE.

3.11.4.19.4. The digraph value is determined in the following way:

3.11.4.19.4.1. The digraphs are identified for each segment.

3.11.4.19.4.2. Each pair of alphabetic characters is identified:
CARA → CA / AR / RA

3.11.4.19.4.3. A digraph is also formed of the initial boundary (#) and the first alphabetic character: CARA → #C.

3.11.4.19.4.4. A digraph is also formed of the final alphabetic character and the final boundary (#): CARA → A#.

3.11.4.19.4.5. The number of shared digraphs is calculated.

3.11.4.19.4.5.1. A digraph may match one digraph only.

3.11.4.19.4.6. The number of shared digraphs is multiplied by 2 and divided by the total number of digraphs in Comparand #1 added to the total number of digraphs in Comparand #2.

3.11.4.19.4.6.1. The formula is:

$$2 * d / a + b,$$

where d = the total number of shared digraphs;

where a = the total number of digraphs in Comparand #1; and

where b = the total number of digraphs in Comparand #2.

3.11.4.19.4.7. The result is the Digraph Value (DI_VAL) for the two Comparands.

Figure 9: Example: Digraph Calculation

COMPARANDS	DIGRAPHS	SHARED DIGRAPHS	DI_VAL
COMPARAND #1: CARA	#C CA AR RA A# (5 total digraphs = a)	#C CA AR A#	$2*d / a + b = 8 / 12$
COMPARAND #2: CARINA	#C CA AR RI IN NA A# (7 total digraphs = b)	= 4 (d)	0.67

3.11.4.19.4.8. This process is performed for each of pair of Comparands.

3.11.4.19.5. To qualify for addition to the HFGV as a variant of one or more HFGN_TYPEs, the digraph value must pass a threshold, the High Frequency Given Name Variant Threshold (HFGV Threshold).

3.11.4.19.5.1. The HFP will access the Hispanic Parameter Data Store (HPD) (Section 4.13) to determine the HFGV Threshold that the digraph value must pass for the LF GN to be appended to the HFGV Data Store.

3.11.4.19.5.2. If the LF GN segment qualifies as digraph variant of one or more HF GN_TYPEs, the HFP

- will append the LF GN to the HFGN_TYPEs to which it is related by entering the name into the HFGN_VAR list in the HFGV Data Store;
- will assign the next available ID_NO to the newly added HFGN_VAR;
- will assign the SET_ID to the newly added HFGN_VAR that corresponds to the SET_ID of the HFGN_TYPE with which the new HFGN_VAR is associated;
- will enter the digraph value into DI_VAL; and
- will store with the LF GN segment in the record the ID_NO(s) of the HFGN_VAR for each entry, the SET_ID of each HFGN_TYPE that is the parent of the HFGN_VAR, and the digraph value associated with each entry.

3.11.4.20. Whether or not the LF segment is a variant of a HFGN_TYPE, the HFP will generate one or more GN_INIT Keys for the LF GN segment.

3.11.4.20.1. The GN_INIT Key will be the initial key for each GN segment, including segments that are initials.

3.11.4.20.2. If the GN segment is FNU (First Name Unknown), no GN_INIT Key will be generated.

3.11.4.20.3. **Record Add**

3.11.4.20.4. The HFP will identify the initial character of each GN segment.

3.11.4.20.5. The HFP will access the Hispanic Character Data Store (HCD) and will find all occurrences of the character in the CHAR-VAR list.

3.11.4.20.6. The HFP will assign the GN_INIT Key(s) to each GN initial.

3.11.4.20.6.1. The GN_INIT Key will be the SET_ID(s) associated with the GN initial (CHAR_VAR).

3.11.4.20.6.2. The GN segment may have multiple GN_INIT Keys.

3.11.4.20.6.3. The GN_INIT Key will permit retrieval of multiple initials for a GN initial.

3.11.4.20.6.4. The HFP will store the GN_INIT Key(s) for each GN segment initial with the record.

3.11.4.20.7. Query

3.11.4.20.8. The HFP will access the HCD and find the initial in the CHAR list.

3.11.4.20.9. The HFP will identify the GN_INIT Key for each GN segment initial.

3.11.4.20.10. The GN_INIT Key will be the SET_ID associated with the CHAR.

Figure 10: Example: HFGN_KEYS and GN_INIT Keys (Query)

INPUT GN	HF?	HFGN_KEY	GN_INIT KEYS
MARIO	T	020	078 (M)
MICHAEL	F		078 (M)
YSABEL	F		036 (Y, I)
ZUSANA	F		002 (Z, S)

Figure 11: Example: HFGN_KEYS and GN_INIT Keys (Record Add)

INPUT GN	HF?	HF VARIANT?	HFGN_TYPE	HFGN_KEY	GN_INIT KEYS
MARIO	T		MARIO	020	078 (M)
MICHAEL	F	F			078 (M)
YSABEL	F	T	ISABEL	203	036 (Y, I)
ZUSANA	F	T	SUSANA	436	002 (Z, S)

3.11.4.21. The HFP will direct queries with all HF SN or mixed HF and LF SN (including SN_INIT Keys) and any GN Keys (HFGN_KEYS or GN_INIT Keys) or FNU to the Hispanic Decision Matrix (HDM) to determine the record retrieval criteria.

3.11.4.21.1. Criteria for database retrieval include name content (whether the names are the same or different), the position of the name segments, the YOB range, the Refusal Code Level, Record Gender and additional restrictions based on the GN.

3.11.4.22. Hispanic Decision Matrix

3.11.4.23. The HFP will access the portion of the HDM that represents the number of HF SN segments in the query name, either one HF SN or two HF SNs.

3.11.4.24. The HFP will identify and generate the set of SN formats possible for the number of SN segments in the query (one or two).

3.11.4.24.1. The SN formats indicate

- position of segments,
- number of segments, and
- other segments permitted.

3.11.4.25. The HFP will identify the retrieval criteria in the HDM associated with each SN format.

3.11.4.25.1. The retrieval criteria include

- Year-of-Birth Range
- Refusal Level and
- Record Gender

3.11.4.26. GN Keys will be carried forward with the retrieval criteria.

3.11.4.27. The HFP will send to the Hispanic Search Engine (HSE) the query format(s), all retrieval criteria associated with each query format and all SN Keys and all GN Keys generated for the query.

Figure 12: Example: Hispanic Decision Matrix (Values for example only)

	Single-Segment SN			Two-Segment SN							
	A	A	A	AB	AB	AB	AB	AB	AB	AB	AB
QUERY SN FORMAT	A	AB	BA	AB	BA	A	B	AC	CA	CB	BC
DATABASE SN FORMATS	5	5	2	5	4	4	2	2	0	0	0
YR	4	4	3	4	4	4	1	1	0	0	0
RL	MFU	MFU	MFU	MFU	MFU	MFU	MFU	FU	MFU	MFU	MFU
RGNDR											

3.11.5. Subordinates

None.

3.12. LOW FREQUENCY PROCESSOR MODULE DECOMPOSITION

3.12.1. Identification

This module is known as the Low Frequency Processor (LFP).

3.12.2. Type

3.12.2.1. The LFP is a program module that will process names that contain SN segments identified by the FPD as Low Frequency SN segments (i.e., not found in the HFST Data Stores by the FPD).

3.12.2.2. The LFP will access the

- High Frequency Surname Variant Data Store (HFSV) and
- Low Frequency Surname Type Data Store (LFST).

3.12.3. Purpose

The LFP will process name segments that are identified as LF SN segments by the FPD. The LFP will determine 1) whether or not the LF segment is a variant of one or more HF SN and 2) whether or not the LF segment has variants among the LF segments listed in the Low Frequency Surname Type Data Store (LFST). The result of these two processes will be a list of segments to use as exact matches for retrieval.

3.12.4. Function

3.12.4.1. General

3.12.4.2. The LFP will accept from the FPD any record with a SN segment that has been tagged as a LF SN.

3.12.4.2.1. The LFP will process records that contain only LF segments *and* all records that contain mixed HF and LF segments.

3.12.4.2.2. The LFP will process records that contain one LF segment and SN_INIT Keys.

3.12.4.2.2.1. Low frequency processing will be limited to the LF segment.

3.12.4.2.2.2. The SN_INIT Keys will contribute to the building of LF retrieval keys.

3.12.4.2.3. With mixed HF and LF SN, the LFP will process only the LF SN segment. (The HF segment will have been assigned a HFSN_KEY by the FPD).

3.12.4.3. All LF segments in records that are sent to the LFP will be analyzed for both HF affiliations and LF variants.

3.12.4.4. The LFP will attempt

- to relate each LF SN segment to one or more HFSN_TYPES,

- to identify other LF SN segments related to the input LF SN segment(s), and
 - to append LF SN segments to the HFSV and LFST that have not been previously submitted to the system to the HFSV and LFST.
- 3.12.4.5. The LFP will direct a query in which all LF SN segment(s) have been related to HFSN_TYPES to the High Frequency Processor (HFP) (see Section 3.11) for generation of GN Keys and submission to the Hispanic Decision Matrix.
- 3.12.4.5.1. The record may have the format HFSN_KEY (or SN_INIT Key) + HFSN_VAR Key, where the second key relates a LF SN segment to a HFSN_TYPE.
- 3.12.4.5.2. The record may have the format HFSN_VAR Key + HFSN_VAR Key, where both segments are keys relating a LF SN segment to a HFSN_TYPE.
- 3.12.4.5.3. The record may have the format HFSN_VAR Key, where the only segment is a key relating a LF SN segment to a HFSN_TYPE.
- 3.12.4.6. The LFP will direct a query record in which all LF SN segments have been related to other LFSN_TYPES directly to the Hispanic Search Engine. (SN_INIT Keys may be present.)
- 3.12.4.7. The LFP will perform the following processes:
- Access the HFSV Data Store to determine if the LF name segment is variant of a HFSN_TYPE.
 - Assign HFSN_VAR Key(s), as appropriate.
 - Generate LF_KEYS for LF SN variants identified in the LFS Data Store.
 - Perform a digraph comparison on the HFST Data Store to determine if a LF SN not in the HFSV Data Store is a digraph variant of a HFSN_TYPE segment.
- 3.12.4.8. The goal of the LFP, for a query with LF SN segments, is to develop a set of specific names related to the LF SN that will be used as keys for record retrieval.
- 3.12.4.9. **Identifying Related High Frequency Surnames**
- 3.12.4.10. The LFP will access the HFSV Data Store to determine if each LF SN segment in the input name is a variant of HFSN_TYPE.
- 3.12.4.10.1. The LFP will attempt to find all occurrences of the LF SN segment in the HFSN_VAR list.
- 3.12.4.10.2. **Record Add**

3.12.4.10.3. If the segment is found in the HFSN_VAR list, the LFP will assign one or more HFSN_KEYs and HFSN_VAR Keys to the LF SN.

3.12.4.10.3.1. The keys will be

- the HFSN_KEY: the SET_ID associated with the HFSN_TYPE that is the parent of the HFSN_VAR and
- the HFSN_VAR Key: the ID_NO of the HFSN_VAR.

3.12.4.10.3.2. The digraph value associated with the HFSN_TYPE and HFSN_VAR pair will be retrieved and stored with the HFSN_KEY and HFSN_VAR Key as the DI_VAL.

3.12.4.10.3.3. The LFP will store the HFSN_KEY, HFSN_VAR Key and the associated digraph value with the record segment.

3.12.4.10.3.3.1. For example, if GARCA is the LF SN and is a variant of the HFSN_TYPE GARCIA, then GARCA will be given the SET_ID associated with the HFSN_TYPE GARCIA (0001) and the ID_NO that uniquely identifies GARCA (000137).

3.12.4.10.3.3.2. The associated digraph value (0.77) will be stored with the LF SN GARCA as the DI_VAL of 0001 and 000137.

3.12.4.10.3.4. There may be multiple HFSN_KEYs and HFSN_VAR Keys associated with a single LF SN segment.

Figure 13: Example: Associating a LF SN Segment with HFSN_TYPE in a Record Add

QUERY SURNAME: PEREZ BOMEZ	HF SN?	HFSN_TYPE	HF KEYS		DI_VAL
			HFSN_KEY	HFSN_VAR KEY	
PEREZ	T	PEREZ	0007		1.00
BOMEZ	F	GOMEZ	0010	016978	0.67

Figure 14: Piece of HFST Data Store

ID_NO	HFSN_TYPE	SET_ID
0001	GARCIA	0001
0002	RODRIGUEZ	0002
0003	HERNANDEZ	0003
0004	LOPEZ	0004
0005	MARTINEZ	0005
0006	GONZALEZ	0006
0007	PEREZ	0007
0008	SANCHEZ	0008
0009	RAMIREZ	0009
0010	GOMEZ	0010
0011	...	0011

Figure 15: Piece of HFSV Data Store

ID_NO	HFSN_VAR	SET_ID	DI_VAL
032711	PEREZ	007	1.00
032712	PERES	007	0.67
032713	PEREZA	007	0.77
016976	GOMEZ	010	1.00
016977	GOMES	010	0.67
016978	BOMEZ	010	0.67

3.12.4.10.4. Query

3.12.4.10.5. The LFP will attempt to associate the LF SN with one or more HFSN_TYPEs.

3.12.4.10.5.1. The LFP will access the HFSV and determine if the LF SN is a variant of a HFSN_TYPE.

3.12.4.10.5.2. If the LF SN is found in the HFSN_VAR list of the HFSV table, the LFP will assign a HFSN_VAR Key to the LF SN segment.

3.12.4.10.5.2.1. The HFSN_VAR Key will be the ID_NO associated with the HFSN_VAR (and *not* the SET_ID that is associated with the HFSN_TYPE).

3.12.4.10.5.2.1.1. The LF segment will be associated with the HF segment but with no other name segments in the same HFSN_TYPE class.

3.12.4.10.5.2.1.2. That is, the variants associated with the HF segment are *not* related to one another through this process.

3.12.4.10.5.3. A LF SN may be a variant of multiple HFSN_TYPES and may therefore receive multiple HFGN_VAR Keys.

3.12.4.10.6. If, by virtue of this process, all LF SN segments in a query are set equal to HFSN_VAR Keys, the LFP will direct the query record to the HFP (see Section 3.11) for generation of Given Name Keys, submission to the High Frequency Decision Matrix (HDM) and identification of retrieval criteria.

Figure 16: Example: Associating a LF SN Segment with HFSN_TYPE in a Query

QUERY SURNAME: PEREZ BOMEZ	HF SN?	HFSN_TYPE	HF KEYS	
			HFSN_KEY	HFSN_VAR KEY
PEREZ	T	PEREZ	0007	
BOMEZ	F	GOMEZ		016978

Figure 17: Piece of HFST Data Store

ID_NO	HFSN_TYPE	SET_ID
0001	GARCIA	0001
0002	RODRIGUEZ	0002
0003	HERNANDEZ	0003
0004	LOPEZ	0004
0005	MARTINEZ	0005
0006	GONZALEZ	0006
0007	PEREZ	0007
0008	SANCHEZ	0008
0009	RAMIREZ	0009
0010	GOMEZ	0010
0011	...	0011

Figure 18: Piece of HFSV Data Store

ID_NO	HFSN_VAR	SET_ID	DI_VALUE
032711	PEREZ	007	1.00
032712	PERES	007	0.67
032713	PEREZA	007	0.77
016976	GOMEZ	010	1.00
016977	GOMES	010	0.67
016978	BOMEZ	010	0.67

3.12.4.10.7. The LFP will direct all queries and record adds to a LF analysis whether or not the LF SN segment was identified as a variant of a HFSN_TYPE.

3.12.4.11. Identifying Related Low Frequency Surnames

3.12.4.12. General

3.12.4.13. All records with one or more LF SN segments will undergo LF analysis by the LFP.

3.12.4.14. For record adds, the LFP will assign an ID_NO that will be stored with the record.

3.12.4.15. The LFP will generate LFDIKEYs for each LF SN segment in the query.

3.12.4.16. The LFP will use the LFDIKEYs to identify related LF SN segments.

3.12.4.17. Note that the LFP will not generate LF Keys for the GN portion of the input name.

3.12.4.18. LF SN Segment in LFST

3.12.4.19. The LFP will determine if the LF SN segment of the input record is in the Low Frequency Surname Type Data Store (LFST).

3.12.4.20. Record Add:

3.12.4.21. If the LF SN segment is a LFSN_TYPE in the LFST, the LFP will assign to and store with the LF SN segment the ID_NO associated with the LFSN_TYPE.

3.12.4.22. Query:

3.12.4.23. If the LF SN segment is a LFSN_TYPE in the LFST, the LFP will retrieve the (up to) 10 digraph keys (LFDIKEYs) that are associated with the name segment in the LFST.

3.12.4.24. The LFP will use the LFDIKEYs retrieved from the LFST and the LFDIKEYs stored with all LFSN_TYPES in the LFST Data to subset the LFST and to identify potential variants of the input LF SN segment.

3.12.4.25. Identifying LF Query Variants

3.12.4.26. Phase 1:

3.12.4.27. The LFP will subset the LFST Data Store.

3.12.4.28. The LFP will select those names from the LFST that share a pre-determined set of LFDIKEYs.

3.12.4.28.1. The LFP will determine the number of LFDIKEYs shared between each LFSN_TYPE and the LF query SN segment.

3.12.4.28.2. The LFP will determine the Shared Key Value based on the number of shared digraphs.

3.12.4.28.2.1. The LFP will use the following formula to determine the Shared Key Value: multiply the number

of shared keys by two and divide by the total number of keys associated with each name:

$$2 * [\text{number of shared keys}] / (\text{total keys of Comparand \#1 plus total keys of Comparand \#2})$$

3.12.4.28.3. The LFP will select only those LFSN_TYPES whose Shared Key Value passes the LFDIKEY Threshold.

3.12.4.28.3.1. The LFP will access the Hispanic Parameter Data Store to identify the minimum matching requirement for the Shared Key Value, the LFDIKEY Threshold.

3.12.4.28.3.2. For a segment to qualify for further processing, the Shared Key Value must pass the LFDIKEY Threshold found in the Hispanic Parameter Data Store (HPD).

Figure 19: Example: Phase 1: LF Variants Related to a LF Query SN Segment; LFDIKEY Threshold = 0.40

	ID_NO		LFDIKEYs	SHARED KEYS	PASS LFDIKEY THRESHOLD 0.40?
QUERY NAME #1		FLORENZAN	FL1 / FL2 / LO2 / LO1 / LO3 / OR3 / OR2 / OR4 / RE4 / RE3		
LFSN_TYPE	000189	FLORENZAN	FL1 / FL2 / LO2 / LO1 / LO3 / OR3 / OR2 / OR4 / RE4 / RE3	10 (All)	2*10/20 = 1.00 YES
LFSN_TYPE	000232	FLORESZ	FL1 / FL2 / LO2 / LO1 / LO3 / OR3 / OR2 / OR4 / RE4 / RE3	10 (All)	2*10/20 = 1.00 YES
LFSN_TYPE	000412	LORENZ	LO1 / LO2 / OR2 / OR1 / OR3 / RE3 / RE2 / RE4 / EN4 / EN3	5 (LO2 / OR2 / OR3 / RE3 / RE4)	2*5/20 = 0.50 YES
QUERY NAME #2		TOREAT	TO1 / TO2 / OR2 / OR1 / OR3 / RE3 / RE2 / RE4 / EA4 / EA3		
LFSN_TYPE	000714	TOREAT	TO1 / TO2 / OR2 / OR1 / OR3 / RE3 / RE2 / RE4 / EA4 / EA3	10 (All)	2*10/20 = 1.00 YES
LFSN_TYPE	000652	THORET	TH1 / TH2 / HO2 / HO1 / HO3 / OR3 / OR2 / OR4 / RE4 / RE3	4 (OR2 / OR3 / RE3 / RE4)	2*4/20 = 0.40 YES
LFSN_TYPE	000776	TOERO	TO1 / TO2 / OE2 / OE1 / OE3 / ER3 / ER2 / ER4 / RO4 / RO3	2 (TO1 / TO2)	2*2/20 = 0.20 NO

3.12.4.29. Phase 2:

3.12.4.30. The LFP will perform a digraph comparison of each LF query SN segment that passed the LFDIKEY Threshold with each LFSN_TYPE.

3.12.4.31. The digraph comparison will identify the set of names to be retrieved from the database.

3.12.4.31.1. See Section 3.11.4.19.4 for the digraph analysis function and formula.

3.12.4.31.2. The LFP will access the Hispanic Parameter Data Store to determine the LF_DI Threshold.

3.12.4.31.3. For a segment to qualify for further processing, the digraph value must pass the LF_DI Threshold found in the Hispanic Parameter Data Store.

Figure 20: Example: Digraph Filter of LFST Candidate SN Segments

LF QUERY SN: FLORENZAN	LFSN_TYPES PASSING LFDIKEY THRESHOLD	DIGRAPH SCORE	PASS LF_DI THRESHOLD: 0.57?
FLORENZAN	FLORENZAN	$2*10/20 = 1.00$	YES
FLORENZAN	FLORESZ	$2*5/18 = 0.56$	NO
FLORENZAN	LORENZ	$2*5/17 = 0.59$	YES

3.12.4.32. The LFP will assign a key (DI_KEY) to each LFSN_TYPE that passes the LF_DI Threshold.

3.12.4.32.1. The DI_KEY will be the ID_NO associated with the LFSN_TYPE in the LFST.

3.12.4.32.2. The DI_KEY will contribute to the building of the retrieval key.

3.12.4.33. For a segment that passes the LF_DI Threshold, the LFP will retain the digraph score derived from the digraph evaluation and associate it with the appropriate DI_KEY.

3.12.4.34. **Low Frequency Surname Segment *Not* in LFST**

3.12.4.35. Add:

3.12.4.36. If the LF SN *record add* segment is **not** in the LFST, the LFP

- will append the LF SN to the LFST as a LFSN_TYPE and assign the next ID_NO available;
- will generate the LFDIKEYs for the new LFSN_TYPE and will add them to the LFST with the LFSN_TYPE;
- will assign the ID_NO to the LF SN segment of added record and
- will determine if the LF SN is a variant of a HFSN_TYPE and therefore should also be added to the HFSV Data Store.

3.12.4.37. The LFP will append the LF SN segment and its LFDIKEYs to the LFST.

3.12.4.37.1. The LFP will assign the next available ID_NO to the newly entered LF SN (LFSN_TYPE).

3.12.4.37.2. The LFP will generate the LFDIKEYs to be associated with the LF SN segment (see 3.13.4.42).

3.12.4.37.3. The up-to-10 keys will be added to the LFST along with the LFSN_TYPE.

3.12.4.37.4. The LFP will assign the LFSN_TYPE ID_NO to the LF SN segment for storage with the record add.

3.12.4.38. The LFP will determine if a LF SN segment that was not identified as a HFSN_VAR in the HFSV *and* that was not identified as a LFSN_TYPE in the LFST is a potential variant of a HFSN_TYPE.

3.12.4.38.1. The LFP will access the High Frequency Surname Type Data Store (HFST) to determine if the LF SN segment is a digraph variant of one or more of the HFSN_TYPES.

3.12.4.38.1.1. The LFP will perform a digraph evaluation of the LF SN and each HFSN_TYPE. (See Section 3.11.4.19.4 for details of the procedure and formula for performing a digraph evaluation.)

3.12.4.38.1.2. To qualify for addition to the HFSV as a variant of one or more HFSN_TYPES, the digraph value must pass a threshold, the High Frequency Surname Variant Threshold (HFSV Threshold).

3.12.4.38.1.3. The LFP will access the Hispanic Parameter Data Store to determine the HFSV Threshold that the digraph value must pass for the LF SN to be appended to the HFSV Data Store.

3.12.4.38.2. If the LF SN segment is determined to be a digraph variant of one or more HFSN_TYPES, the LFP

- will append the LF SN to the HFSN_TYPES to which it is related by entering the name into the HFSN_VAR list;
- will assign an ID_NO to the newly added HFSN_VAR;
- will assign the SET_ID to the newly added HFSN_VAR that corresponds to the SET_ID of the HFSN_TYPE with which the new HFSN_VAR is associated;
- will enter the digraph value into DI_VAL; and
- will store with the LF SN segment in the record add the ID_NO of the HFSN_VAR for each entry, the SET_ID of each HFSN_TYPE that is the parent of the HFSN_VAR and the DI_VAL for each relationship.

3.12.4.39. Query

3.12.4.40. If the LF SN *query* segment is not in the LFST, the LFP

- will generate the LFDIKEYs for the new LF SN;
- will select the related LFSN_TYPES through the LF selection process; and
- will determine if the LF SN is a variant of a HFSN_TYPE, assign appropriate keys and retain related digraph values. (See Section 3.12.4.43).

- 3.12.4.41. The LFP will generate the LFDIKEYs for the LF SN segment (see Section 3.12.4.44).
- 3.12.4.42. The LFP will identify LFSN_TYPEs in the LFST that are variants of the LF. (See Section 3.12.4.11 for the identification process.)
- 3.12.4.43. The LFP will determine if a LF SN segment that was not identified as a HFSN_VAR *and* that was not found in the LFST is a potential variant of a HFSN_TYPE.
 - 3.12.4.43.1. The LFP will access the HFST Data Store and perform a digraph evaluation between the LF SN and each HFSN_TYPE. (See Section 3.11.4.19.4 for details of the procedure and formula for performing a digraph evaluation.)
 - 3.12.4.43.2. The digraph value must pass a threshold for the LF SN to be considered a variant of a HFSN_TYPE(s), the High Frequency Surname Variant Threshold (HFSV Threshold).
 - 3.12.4.43.3. The LFP will access the Hispanic Parameter Data Store to determine the HFSV Threshold that the digraph value must pass for the LF SN to qualify as a variant of a HFSN_TYPE.
 - 3.12.4.43.4. If the LF SN segment passes the HFSV Threshold, the LFP will assign HFSN_VAR Key(s) to the LF SN segment.
 - 3.12.4.43.4.1. The HFSN_VAR Key will be the ID_NO associated with the HFSN_VAR that is equal to the HFSN_TYPE.
 - 3.12.4.43.4.1.1. That is, the LF SN segment will be associated with the parent HFSN_TYPE only.
 - 3.12.4.43.4.1.2. A LF SN may be a variant of multiple HFSN_TYPEs and may therefore receive multiple HFSN_VAR Keys.
 - 3.12.4.43.4.2. The calculated digraph value will be associated with each HFSN_VAR Key.
 - 3.12.4.43.5. If, by virtue of this process, all LF SN segments in a query are set equal to HFSN_VAR Keys, the LFP will direct the query record to the HFP (Section 3.11.) for generation of Given Name Keys, submission to the High Frequency Decision Matrix (HDM) and identification of retrieval criteria.

3.12.4.44. Generating LFDIKEYs

- 3.12.4.44.1. The LFDIKEY is
 - 1) a set of digraphs formed from the LF SN segment beginning with the leftmost character and
 - 2) a set of positional variants on those digraphs.

3.12.4.44.2. Positional information will be associated with each digraph.

3.12.4.44.3. Base Keys

3.12.4.44.4. The LFP will begin with the leftmost character and generate up to four digraph keys (Base Keys) from the (up to) five leftmost characters of the LF SN segment.

3.12.4.44.4.1. The first two characters form a digraph, the second and third characters form a digraph, the third and fourth characters form a digraph and the fourth and fifth characters form a digraph.

3.12.4.44.4.2. Positional information will be included: 1, 2, 3, 4, respectively: DI1, DI2, DI3, DI4.

3.12.4.44.5. If the LF SN segment has fewer than five characters, the LFP will generate fewer than four Base Keys, up to the number of characters in the LF SN.

3.12.4.44.6. Positional information will be included.

3.12.4.44.7. Position Keys

3.12.4.44.8. The LFP will generate from the Base Keys up to six additional Position Keys from the Base Keys.

3.12.4.44.8.1. A maximum of ten keys (Base + Position) will be generated.

3.12.4.44.8.2. The Position Keys have the same characters as the Base Keys but contain different positional information.

3.12.4.44.8.3. For segments with 5 or more characters:

3.12.4.44.8.4. The LFP will produce a Position Key on the first Base Key with Position 2.

3.12.4.44.8.5. The LFP will produce Position Keys on the second Base Key with Position 1 and Position 3.

3.12.4.44.8.6. The LFP will produce Position Keys on the third Base Key with Position 2 and Position 4.

3.12.4.44.8.7. The LFP will produce a Position Key on the fourth Base Key with Position 3. No Position Key is generated for Position 5 because the maximum of 10 keys has been reached.

3.12.4.44.8.8. For segments with fewer than 5 characters:

3.12.4.44.8.9. The LFP will produce Position Keys in the same way as for longer LF SN segments.

- 3.12.4.44.8.9.1. No Position Key will be generated for the final Base Key with a position to the right of the final character.
- 3.12.4.44.8.9.2. The total number of LFDIKEYs will be fewer than with a longer LF SN segment.
- 3.12.4.44.8.9.3. In GOMA, the LFP will generate a total of 7 keys: the Base Keys GO1, OM2 and MA3, and the Position Keys, GO2, OM1, OM3, and MA2. (Note: No MA4 Position Key is produced for the final digraph.)

Figure 21: Example: LFDIKEYs for LF SN Segments

LF SN SEGMENT	LFDIKEYS: BASE KEYS	LFDIKEYS: POSITION KEYS
CARRIOS	CA1 / AR2 / RR3 / RI4	CA2 / AR1 / AR3 / RR2 / RR4 / RI3
BALA	BA1 / AL2 / LA3	BA2 / AL1 / AL3 / LA2

3.12.4.44.9. Building LF Retrieval Keys (Query)

3.12.4.44.10. General

- 3.12.4.44.11. Each LF SN segment has been assigned a DI_KEY or set of DI_KEYs.
- 3.12.4.44.12. A LF SN segment may also have been assigned one or more HFSN_VAR Keys.
- 3.12.4.44.13. The LFP has sent queries with all HFSN_KEYs and/or HFSN_VAR Keys (including SN_INIT Keys) to the HFP for further processing.
- 3.12.4.44.14. The LFP will build sets of retrieval keys for mixed frequency queries (at least one HF key and one LF SN key in the string) and for queries with all low frequency keys (all SN in the string must be DI_KEYs).
 - 3.12.4.44.14.1. A single query may have various formats – all HF, mixed and/or all-LF SN depending on the results of LF processing prior to this stage.
- 3.12.4.44.15. Mixed frequency queries will not contain SN_INIT Keys.
 - 3.12.4.44.15.1. SN_INIT Keys may occur with HF SN keys, in which case the record will be treated as an all-HF SN record.
 - 3.12.4.44.15.2. SN_INIT Keys may occur with LF SN keys, in which case the record will be treated as an all-LF SN record.

3.12.4.44.16. Queries with Mixed Frequency (HF + LF) Surnames

3.12.4.44.17. Type 1:

3.12.4.44.18. If one SN in the query is a HF SN and has an associated HFSN_KEY and one SN in the query record is a LF SN segment and has associated DI_KEYs, the LFP will build a Mixed Key of the HFSN_KEY and each DI_KEY (and the associated DI_VALs).

3.12.4.44.18.1. The HFSN_KEY represents a set of variants of one HFSN_TYPE.

3.12.4.44.18.1.1. GARCIA, GARCA, GARZA are all digraph variants of the HFSN_TYPE GARCIA, which has the SET_ID 0001.

3.12.4.44.18.1.2. Record adds and queries will already have been assigned the HFSN_KEY through the HFP.

3.12.4.44.18.2. The DI_KEY represents a *single* low frequency surname type that has qualified through the LF SN selection process.

Figure 22: Example: Building Mixed HF/LF SN Retrieval Keys with HFSN_KEY and DI_KEYs

QUERY NAME: GARCIA FLORENZAN	HFSN_KEY and (LF) DI_KEY
GARCIA (HF)	GARCIA → 001
FLORENZAN (LF)	FLORENZAN → 000189
GARCIA (HF)	GARCIA → 001
LORENZ (LF)	LORENZ → 000412

3.12.4.44.19. Type 2:

3.12.4.44.20. If one SN in the query record is a HF SN and has an associated HFSN_VAR Key (generated by the LFP) and one SN in the query record is a LF SN segment with associated DI_KEY(s), the LFP will build a Mixed Key of the HFSN_VAR Key and the DI_KEY for each qualifying LFSN_TYPE.

3.12.4.44.20.1. The HFSN_VAR represents a *single* HFSN_TYPE and *not* a set of variants.

3.12.4.44.20.1.1. Record adds and queries will already have been assigned the HFSN_VAR Key through the LFP.

3.12.4.44.20.2. The DI_KEY represents a *single* low frequency surname type that has qualified through the LF SN selection process.

Figure 23: Example: Building Mixed HF/LF SN Retrieval Keys with HFSN_VAR Keys and DI_KEYS

QUERY NAME: GARCIA FLORENZAN	HFSN_VAR and (LF) DI_KEY
BOMEZ (LF → HFSN_VAR)	BOMEZ → 016978
FLORENZAN (LF)	FLORENZAN → 000189
BOMEZ (LF → HFSN_VAR)	BOMEZ → 016978
LORENZ (LF)	LORENZ → 000412

3.12.4.44.21. It is likely that there will be multiple DI_KEYS for each LF SN segment, resulting in multiple Mixed Keys.

3.12.4.44.22. Once the Mixed SN Keys have been generated (and DI_VALs associated with the appropriate keys), the LFP will send any query that contains mixed HF and LF Keys to the HFP (Section 3.11.4.5) for Given Name processing and identification of retrieval criteria from the Hispanic Decision Matrix.

3.12.4.44.23. **Queries with All Low Frequency (LF + LF) Surnames**

3.12.4.44.24. The LFP will identify the LF Keys associated with query formats made up solely of LF SN segments (or a LF SN segment and SN_INIT Key(s)).

3.12.4.44.24.1. The LFP has qualified one or more LF SN segments from the LFST as variants of each LF query SN.

3.12.4.44.24.2. Each qualifying LF segment has been assigned a LF Key, the DI_KEY, and has an associated digraph value, DI_VAL.

Figure 24: Example: Low Frequency DI_KEYS and Associated Digraph Values

QUERY NAME: TOREAT FLORENZAN	LFST ID_NO	DI_KEYS + DI_VAL
TOREAT	TOREAT → 000714	000714 (1.00)
THORET	THORET → 000652	000652 (0.57)
FLORENZAN	FLORENZAN → 000189	000189 (1.00)
FLORESZ	FLORESZ → 000232	000232 (0.56)
LORENZ	LORENZ → 000412	000412 (0.59)

3.12.4.44.25. The LFP will direct a query record with all DI_KEYS and their digraph values (or DI_KEYS and SN_INIT Keys) to the Hispanic Search Engine for retrieval of database records.

3.12.5. Subordinates

None.

3.13. HISPANIC SEARCH ENGINE MODULE DECOMPOSITION

3.13.1. Identification

This module is known as the Hispanic Search Engine (HSE).

3.13.2. Type

3.13.2.1. The HSE is a function that applies to queries only.

3.13.2.2. The HSE will accept name keys and retrieval criteria from the HFP and the LFP.

3.13.2.3. The module must follow the HFP and LFP.

3.13.3. Purpose

The HSE will retrieve records from the VLDB based on criteria identified by the High Frequency Processor and the Low Frequency Processor. These criteria will delimit the set of records that can qualify for retrieval. The system must be sure that the criteria have all been identified and can be associated with database records (whether through database design and/or key generation).

3.13.4. Function

3.13.5. HNA-E will not handle records with Last Name Unknown (LNU).

3.13.6. The HSE will permit First Name Unknown (FNU).

3.13.6.1. The processing of FNU will supersede other GN restrictions.

3.13.6.2. The HSE will retrieve any database GN when FNU occurs in the query.

3.13.6.3. The HSE will retrieve any FNU in the database for any query GN.

3.13.7. High Frequency Retrieval

3.13.7.1. High frequency retrieval will include records with HFSN_KEYS, HFSN_VAR Keys and SN_INIT Keys that occur with the HF SN keys.

3.13.7.1.1. The SN_INIT Key will result in the retrieval of records that begin with or are equal to the variant initials identified by the SN_INIT Key.

3.13.7.1.2. The SN_INIT Key is stored with each SN segment.

3.13.7.1.3. All HF retrieval restrictions apply to the SN_INIT Key, as if it were a HF segment *except* that

3.13.7.1.3.1. If the SN_INIT Key is the only key in the format, the HSE will not undertake a database search.

3.13.7.1.4. No further, separate detailing of the SN_INIT Key is given.

3.13.7.2. High frequency retrieval will include mixed HF and LF SN Keys but with no SN_INIT Keys.

3.13.8. All HFSN_KEYS (or SN_INIT Key)

3.13.9. For queries with all HFSN_KEYS, the HSE will retrieve records from the database records that

- contain the appropriate SN format (position, more/fewer segments, different segments) as specified in the HDM,
- contain the appropriate HFSN_KEYS,
- meet all the criteria identified in the HDM and
- meet the GN restrictions.

3.13.10. The HFSN_KEY will result in retrieval of the HFSN_TYPE and all its variants.

3.13.10.1. The HSE will further restrict the retrieval to records that match at least one key of the GN.

3.13.10.1.1. If the query has produced only HFGN_KEYS, only records that have at least one of the HFGN_KEYS will be retrieved.

3.13.10.1.2. If the query has produced mixed HFGN_KEYS and GN_INIT Keys, the HSE will retrieve records that match at least one of the HFGN_KEY or one of the GN_INIT Keys.

3.13.10.1.3. If the query has produced only GN_INIT Keys, the HSE will retrieve records that match at least one of the GN_INIT Keys.

Figure 25: Example: Record Matching Criteria: All HFSN_KEYS and HFGN_KEYS

QUERY #1	RODRIGUEZ	LOPEZ	JOSE	CARLOS	CRITERIA
HFSN_KEY	002	010			
HFGN_KEY			0001	0007	
HDM FORMATS:					
1	RODRIGUEZ (002)	LOPEZ (010)			YOB5, RL4, MFU, GN contains 0001 or 0007
2	LOPEZ	RODRIGUEZ			YOB4, RL4, MFU, GN contains 0001 or 0007
3	RODRIGUEZ				YOB4, RL4, MFU, GN contains 0001 or 0007
4	LOPEZ				YOB2, RL1, MFU, GN contains 0001 or 0007
5	RODRIGUEZ	* (ANY SN)			YOB2, RL1, FU, GN contains 0001 or 0007
6	LOPEZ	* (ANY SN)			YOB0, RL0, MFU, GN contains 0001 or 0007
7	* (ANY SN)	RODRIGUEZ			YOB0, RL0, MFU, GN contains 0001 or 0007
8	* (ANY SN)	LOPEZ			YOB0, RL0, MFU, GN contains 0001 or 0007

Figure 26: Example: Record Matching Criteria: All HFSN_KEYS and Mixed HFGN_KEYS and GN_INIT Keys

QUERY #2	RODRIGUEZ	LOPEZ	JOSSE	CARLOS	CRITERIA
HFSN_KEY	002	010			
HFGN_KEY				0007	
GN_INIT Key(s)			041 (J, H)		
HDM FORMATS:					
1	RODRIGUEZ (002)	LOPEZ (010)			YOB5, RL4, MFU, GN initial = J or H; or GN= 0007
2	LOPEZ	RODRIGUEZ			YOB4, RL4, MFU, GN initial = J or H; or GN= 0007
3	RODRIGUEZ				YOB4, RL4, MFU, GN initial = J or H; or GN= 0007
4	LOPEZ				YOB2, RL1, MFU, GN initial = J or H; or GN= 0007
5	RODRIGUEZ	*			YOB2, RL1, FU, GN initial = J or H; or GN= 0007
6	LOPEZ	*			YOB0, RL0, MFU, GN initial = J or H; or GN= 0007
7	*	RODRIGUEZ			YOB0, RL0, MFU, GN initial = J or H; or GN= 0007
8	*	LOPEZ			YOB0, RL0, MFU, GN initial = J or H; or GN= 0007

3.13.11. HFSN_KEY and/or HFSN_VAR Keys (or SN_INIT Key)

3.13.12. For queries with mixed HFSN_KEYS and HFSN_VAR Keys and queries with all HFSN_VAR Keys, the HSE will retrieve records from the database records that

- contain the appropriate HFSN_KEYS and/or HFSN_VAR Keys,

- contain the appropriate SN format (position, more/fewer segments, different segments) as specified in the HDM,
- meet all the criteria identified in the HDM and
- meet the GN restrictions.

3.13.13. The HFSN_VAR Key will retrieve a *single* HFSN_TYPE and not the set of variants associated with the HFSN_TYPE (e.g., the name LOPEZ but not all its variants; the variants will be retrieved by the LFP).

3.13.13.1. The HSE will further restrict the retrieval to records that match at least one key of the GN.

3.13.13.1.1. If the query has produced only HFGN_KEYs, only records that have at least one of the HFGN_KEYs will be retrieved.

3.13.13.1.2. If the query has produced mixed HFGN_KEYs and GN_INIT Keys, the HSE will retrieve records that match at least one of the HFGN_KEY or the GN_INIT Keys.

3.13.13.1.3. If the query has produced only GN_INIT Keys, the HSE will retrieve records that match at least one of the GN_INIT Keys.

Figure 27: Example: Record Matching Criteria: All HFSN_KEY and/or HFSN_VAR Keys

QUERY #1	RODRIGUEZ	SLOPEZ	JOSSE	CARLOS	CRITERIA
HFSN_KEY	002				
HFSN_VAR Key		00976			
HFGN_KEY				0007	
GN_INIT Key(s)			041 (J, H)		
HDM FORMATS:					
1	RODRIGUEZ (002)	LOPEZ (000976)			YOB5, RL4, MFU, GN initial = J or H; or GN= 0007
2	LOPEZ	RODRIGUEZ			YOB4, RL4, MFU, GN initial = J or H; or GN= 0007
3	RODRIGUEZ				YOB4, RL4, MFU, GN initial = J or H; or GN= 0007
4	LOPEZ				YOB2, RL1, MFU, GN initial = J or H; or GN= 0007
5	RODRIGUEZ	*			YOB2, RL1, FU, GN initial = J or H; or GN= 0007
6	LOPEZ	*			YOB0, RL0, MFU, GN initial = J or H; or GN= 0007
7	*	RODRIGUEZ			YOB0, RL0, MFU, GN initial = J or H; or GN= 0007
8	*	LOPEZ			YOB0, RL0, MFU, GN initial = J or H; or GN= 0007

3.13.14. Mixed HFSN_KEY and/or HFSN_VAR Keys and LF DI_KEYS (no SN_INIT Key)

3.13.15. For queries with mixed HFSN_KEYS/HFSN_VAR Keys and LF DI_KEYS, the HSE will retrieve records from the database records that

- contain the appropriate HFSN_KEYS/HFSN_VAR Keys and DI_KEYS,
- contain the appropriate SN format (position, more/fewer segments, different segments) as retrieved from the HDM,
- meet all the criteria identified in the HDM and
- meet the GN restrictions.

3.13.16. The LFP generated (multiple) query formats that contain a HF Key and a DI_KEY.

3.13.16.1. The DI_KEY will retrieve an exact match on a *single* LFSN_TYPE.

3.13.16.2. Each HFSN_KEY or HFSN_VAR Key may participate in query formats with several different DI_KEYS that were identified as variants by the LFP.

3.13.16.3. Each query format will serve as a different query.

3.13.17. The HSE will further restrict the retrieval to records that match at least one key of the GN.

3.13.17.1. If the query has produced only HFGN_KEYS, only records that have at least one of the HFGN_KEYS will be retrieved.

3.13.17.2. If the query has produced mixed HFGN_KEYS and GN_INIT Keys, the HSE will retrieve records that match at least one of the HFGN_KEY or the GN_INIT Keys.

3.13.17.3. If the query has produced only GN_INIT Keys, the HSE will retrieve records that match at least one of the GN_INIT Keys.

Figure 28: Example: Record Matching Criteria: Mixed HFSN_KEYS/HFSN_VAR Keys and LFDI_KEYS

QUERY #1	THORET	SLOPEZ	JOSSE	CARLOS	CRITERIA
HFSN_KEY					
HFSN_VAR Key		00976			
DI_KEY	000652 (THORET)				
	000714 (TOREAT)				
HFGN_KEY				0007	
GN_INIT Key(s)			041 (J, H)		
HDM FORMATS:					
1	THORET	LOPEZ (000976)			YOB5, RL4, MFU, GN initial = J or H; or GN= 0007
2	LOPEZ	THORET			YOB4, RL4, MFU, GN initial = J or H; or GN= 0007
3	THORET				YOB4, RL4, MFU, GN initial = J or H; or GN= 0007
4	LOPEZ				YOB2, RL1, MFU, GN initial = J or H; or GN= 0007
5	THORET	*			YOB2, RL1, FU, GN initial = J or H; or GN= 0007
6	LOPEZ	*			YOB0, RL0, MFU, GN initial = J or H; or GN= 0007
7	*	THORET			YOB0, RL0, MFU, GN initial = J or H; or GN= 0007
8	*	LOPEZ			YOB0, RL0, MFU, GN initial = J or H; or GN= 0007
1	TOREAT	LOPEZ			YOB5, RL4, MFU, GN initial = J or H; or GN= 0007
2	LOPEZ	TOREAT			YOB4, RL4, MFU, GN initial = J or H; or GN= 0007
3	TOREAT				YOB4, RL4, MFU, GN initial = J or H; or GN= 0007
4	LOPEZ				YOB2, RL1, MFU, GN initial = J or H; or GN= 0007
5	TOREAT	*			YOB2, RL1, FU, GN initial = J or H; or GN= 0007
6	LOPEZ	*			YOB0, RL0, MFU, GN initial = J or H; or GN= 0007
7	*	TOREAT			YOB0, RL0, MFU, GN initial = J or H; or GN= 0007
8	*	LOPEZ			YOB0, RL0, MFU, GN initial = J or H; or GN= 0007

3.13.18. The HSE will not retrieve database records that have already been retrieved with another key.

3.13.19. The HSE will retrieve the database record ID; the Dual-SN Formats; keys, their segment position and their related DI_VALs; Record Gender; and TAQ tags.

3.13.20. Low Frequency Retrieval

3.13.21. The HSE will retrieve records from the database that contain one or both of the query DI_KEYs in any SN position in the database record.

3.13.21.1. The HSE will retrieve records that contain the DI_KEYs within a specified YOB Range for a Refusal Code Level.

3.13.21.2. The HSE will access the RLYOB Data Store to determine the Refusal Code Level and associated Year-of-Birth Range that will apply.

3.13.21.3. The HSE will retrieve all records from the database with

- both DI_KEYs (or one DI_KEY and one SN_INIT Key) in either position and RLYOB restriction;
- one of the DI_KEYs alone and RLYOB restriction; and
- one DI_KEY in either position, if the Year-of-Birth Range is YOB2 and the Refusal Code Level is RL1 (i.e., 00 or Type 1 Serious).

3.13.22. The HSE will retrieve the database record; the record ID; the Dual-SN Formats; keys, their segment position and their related DI_VALs; Record Gender; and TAQ tags.

3.13.23. The HSE will not retrieve a record that has already been retrieved using other access methods (i.e., Mixed Frequency SN or HF names).

3.13.24. All records retrieved from the database will be sent to the Hispanic Filter and Sorter.

3.14. HISPANIC FILTER AND SORTER MODULE DECOMPOSITION

3.14.1. Identification

This module is known as the Hispanic Filter and Sorter (HFS).

3.14.2. Type

3.14.2.1. The HFS is a module that accepts database records retrieved by the HSE.

3.14.2.2. The HFS compares each database record to the query record to determine if it qualifies for return to the user.

3.14.2.3. The HFS is constituted of two subordinate functions: the Hispanic Filter and the Hispanic Sorter.

3.14.2.4. The HFS must follow the Hispanic Search Engine (HSE).

3.14.3. Purpose

3.14.3.1. The set of database records that the HSE will retrieve will be a set of records delimited by quite narrow retrieval criteria. The database records will have a digraph value associated with most SN segments and with many GN segments. However, the relative value of the database records to the query record will not be clear. The HFS will, therefore, evaluate each of the records retrieved for its proximity to a query record, will retain those that pass a pre-established threshold and will sort the resultant candidate list.

3.14.3.2. The filtering process will take into account a number of factors that play a role in determining the relative value of Hispanic *names*.

3.14.3.3. The filtering process will take into account factors that aid in the determination of the relative value of a Hispanic *records*.

3.14.4. Function

The HFS will first compare and qualify the query name and database-record name to determine a surname value (SN_VAL), will then evaluate and qualify the query name and database record to determine a given name value (GN_VAL) and will generate a composite score for the database records that qualified on the basis of name evaluation by factoring in values for Date-of-Birth, Refusal Level and Country of Birth.

The first comparison will be to identify an exact record match. All other comparison will be between the Dual-SN Format of the query and database record (for records with more than two surnames).

3.14.4.1. Filter Function of the HFS

3.14.4.2. General

3.14.4.3. The Hispanic Filter and Sorter (HFS) will accept the candidate database records retrieved by the HSE.

3.14.4.4. The HFS will first determine if the query record and database record match exactly.

3.14.4.4.1. The HFS will compare the base format of the query and database record; i.e., no derived format.

3.14.4.4.2. The name (both SN and GN), Date-of-Birth and Country-of-Birth must match exactly.

3.14.4.4.3. If the query and database records match exactly, the HFS will tag the record as an exact match and send the record directly to the Sorter Function of the HFS.

3.14.4.5. The HFS will calculate name scores for each candidate database record as it compares to the query record.

3.14.4.5.1. The HFS will use the derived formats as the basis of record comparison.

3.14.4.5.2. A score for the SN, the SN_VAL, will be calculated.

3.14.4.5.3. A score for the GN, the GN_VAL, will be calculated.

3.14.4.5.3.1. The HFS will adjust the digraph value retrieved with the database record by multiplying that value by factors assigned to several parameters.

3.14.4.5.3.2. Factors (see Section 4.13) that contribute to the determination and evaluation of the name score (SN_VAL and GN_VAL) include

- SNTHR
- GNTHR
- ASVAL
- AGVAL
- OPSVAL
- OPGVAL
- INTSN
- INITGN
- TAQASN
- TAQAGN
- TAQXSN
- TAQXGN
- RGNDR

3.14.4.5.3.3. To be included in the final candidate list, the score of the SN and the score of the GN must each pass pre-determined SN and GN threshold levels (SNTHR and GNTHR).

3.14.4.6. Surname Evaluation

3.14.4.7. A candidate record must pass a SN evaluation before it will be submitted to a GN evaluation.

3.14.4.8. No record with Last Name Unknown (LNU) will be handled by HNA-E.

3.14.4.9. The SN evaluation will be performed on the derived formats (including the Dual-SN Formats) associated with the query and database records.

3.14.4.10. High Frequency SN Keys (HFSN_KEYS or HFSN_VAR Keys)

3.14.4.10.1. The HFS will compare the keys of the query and database and assign the DI_VAL retrieved with the database record to the SN Comparands with matching keys.

3.14.4.10.1.1. Only one assignment of DI_VAL can be made for a match.

3.14.4.10.1.2. If the query is GARCIA GOMEZ and the database record is GARCIA GARCIA, the HFS will assign the DI_VAL to one GARCIA match only.

3.14.4.10.2. If the SN Keys do not match, the HFS will perform a digraph match of the segments with no assigned value (LOPEZ and GOMEZ in Figure 29) and will assign the digraph score to the DI_VAL.

Figure 29: Example: Database Records with HFSN_KEYS to be Evaluated by HFS

	SN#1	HFSN_KEY	DI_VAL	SN#2	HFSN_KEY	DI_VAL
QUERY	GARCIA	0001		GOMEZ	0010	
DATABASE RECORDS	GARCIA	0001	1.00	BOMEZ	0010	0.67
	BARCIA	0001	0.71	GAMEZ	0010	0.67
	LOPEZ	0004	0.17	GARCIA	0001	1.00

3.14.4.11. Low Frequency SN Keys (DI_KEYS)

3.14.4.11.1. The HFS will assign the DI_VAL associated with the DI_KEY to matching database and query DI_KEYS.

3.14.4.11.1.1. Only one assignment of DI_VAL can be made for a match.

- 3.14.4.11.1.2. If the query is THORET FLORENZAN and the database record is THORET THORET, the HFS will assign the DI_VAL to one THORET match only.
- 3.14.4.12. If the SN Keys do not match, the HFS will perform a digraph match of the segments with no assigned value (LOPEZ and GARCIA in Figure 30) and will assign the digraph score to the DI_VAL.
- 3.14.4.12.1. If there is more than one pair that does not have an assigned digraph value, the HFS will perform a digraph evaluation for each of the pairs. (See Section 3.14.4.16 for details of the digraph assignment.)
- 3.14.4.12.2. Each value will be submitted to parameter evaluation.
- 3.14.4.12.3. After all parameters have been applied, the HFS will choose the highest score for each pair. (See Section 3.14.4.17)

Figure 30: Example: Database Records with LF SN (Mixed or all LF Keys) to be Evaluated by HFS

	SN#1	HFSN_KEY/ DI_KEY	DI_VAL	SN#2	HFSN_KEY/ DI_KEY	DI_VAL
QUERY	GARCIA	0001		THORET	000652	
DATABASE RECORDS	GARCIA	0001	1.00	THORET	000652	1.00
	THORET	000652	1.00	BARCIA	0001	0.71
	LOPEZ	0004	0.00	THORET	000652	1.00

- 3.14.4.13. The HFS will adjust the DI_VAL of each segment according to parameter values in the Hispanic Parameter Data Store (see Section 4.13 for details).
- 3.14.4.13.1. The HFS will determine if the appropriate parameter conditions obtain.
- 3.14.4.13.2. If the appropriate conditions are present, the DI_VAL will be multiplied by the value assigned to the parameter and the DI_VAL will be lowered.
- 3.14.4.13.3. **Parameter Conditions**
- 3.14.4.13.4. **INITSN: Initial**
- 3.14.4.13.4.1. Definition 1: The SN segment is a single character in both comparands and the character matches exactly.
- 3.14.4.13.4.2. Action: The HFS will make no change.
- 3.14.4.13.4.3. Definition 2: A SN segment is a single character and its SN_INIT Key matches the SN_INIT Key of the other comparand.

3.14.4.13.4.4. Action: Assign the INITSN value to the comparison value (i.e., do not calculate the DI_VAL). The initial may be subjected to any following actions (e.g., out-of-place segment).

3.14.4.13.4.5. Definition 3: A SN segment is a single character and the SN_INIT Keys of the comparands do not match.

3.14.4.13.4.6. Action: Assign the INITNM value to the comparison value (i.e., do not calculate the DI_VAL). The initial may be subjected to any following actions (e.g., out-of-place segment).

3.14.4.13.5. OPSVAL: Out-of-Place Surname Value

3.14.4.13.5.1. Definition: A SN segment that is not in the same relative position in the SN string in both the database and query records.

3.14.4.13.5.2. Action: Multiply the DI_VAL by the OPSVAL.

3.14.4.13.6. ASVAL: Anchor Surname Value

3.14.4.13.6.1. Definition: For database records that contain two SN segments, the database SN segments are in the correct position relative to the query SN segments.

3.14.4.13.6.2. Action: Multiply the DI_VAL of the second (rightmost) segment by the ASVAL.

Figure 31: Example 1: SN Parameter Evaluation: OPSN Applies

	GARCIA	GOMEZ
BOMEZ		$0.67 * 0.65 = 0.44$
GARCIA	$1.00 * 0.65 = 0.65$	

Figure 32: Example 2: SN Parameter Evaluation: OPSN Applies

	GARCIA	GOMEZ
GAMEZ		$0.67 * 0.65 = 0.44$

Figure 33: Example 3: SN Parameter Evaluation: ASVAL Applies

	GARCIA	GOMEZ
GARZA	0.62	
GOMEZ		$1.00 * 0.65 = 0.65$

3.14.4.13.7. TAQ Filter

3.14.4.13.8. All TAQ tags (ID_NO, disposition, TAQ_TYPE and associated SN stem) will be retrieved with the database record.

3.14.4.13.9. The HFS will evaluate any TAQs associated with the SN segments being evaluated, except Stranded Prefixes (see Section 3.5.4.2.7.3).

3.14.4.13.9.1. A Stranded Prefix will not play a role in the record comparison.

3.14.4.13.10. **Single TAQs**

3.14.4.13.11. **Missing TAQs**

3.14.4.13.12. **TAQASN: Absent TAQ Value**

3.14.4.13.12.1. Definition 1: One of the two comparands (query/database SN segment) has a TAQ tag, the other does not.

3.14.4.13.12.2. Definition 2: Both comparands (query/database SN segments) have a single TAQ tag, one is a TAQ DELETE, the other a TAQ DISREGARD.

3.14.4.13.12.3. Action: Multiply the DI_VAL by the TAQASN value.

Figure 34: Example: TAQ DISREGARD (DE) and No TAQ

	DE VARGAS
VARGAS	$1.00 * 0.90 = 0.90$

Figure 35: Example: TAQ DISREGARD (DE) and TAQ DELETE (DR)

	DE VARGAS
DR VARGAS	$1.00 * 0.90 = 0.90$

3.14.4.13.13. **TAQ DELETE**

3.14.4.13.14. **TAQXSN: Deleted TAQ Value**

3.14.4.13.14.1. Definition: Both SN comparands have a single TAQ DELETE tag.

3.14.4.13.14.2. Action:

3.14.4.13.14.3. If the TAQ DELETE tags refer to the same TAQ segment, the DI_VAL will be unchanged.

3.14.4.13.14.4. If the TAQ DELETE tags refer to different TAQ DELETE segments, multiply the DI_VAL by the TAQXSN value.

Figure 36: Example: Same TAQ DELETE (DR)

	DR VARGAS
DR VARGAS	1.00

Figure 37: Example: Different TAQ DELETES (DR and SR)

	SR VARGAS
DR VARGAS	$1.00 * 0.850 = 0.85$

3.14.4.13.15. TAQ DISREGARD

3.14.4.13.15.1. Definition: The HFS will access the TAQ Filter Data Store (TF) to process records that both contain SN TAQ segments that have been tagged as DISREGARD.

3.14.4.13.15.2. Action 1: The HFS will assign TAQDIS#1 to the TAQ DISREGARD segment for the database SN segment and TAQDIS#2 to the TAQ DISREGARD segment for the query SN segment.

3.14.4.13.15.3. Action 2: If the two TAQ DISREGARD segments match, the DI_VAL will remain unchanged.

3.14.4.13.15.4. Action 3: If the two TAQ DISREGARD segments do not match, the HFS will identify the TF_VALUE for the pair in the TF.

3.14.4.13.15.4.1. The HFS will multiply the DI_VAL by the TF_VALUE for the pair.

Figure 38: Example: Different TAQ DISREGARDS (DE and LA)

	DE PENA
LA PENA	$1.00 * 0.75 = 0.75$

3.14.4.13.16. Multipart TAQs

3.14.4.13.16.1. Definition: If at least one SN comparand has multipart TAQ tags (they may be all DISREGARD, all DELETE, or mixed DISREGARD/DELETE), the HFS will perform the following analyses.

3.14.4.13.16.2. Action: If all TAQs match, HFS will make no change in the DI_VAL.

3.14.4.13.16.3. TAQ DELETES

3.14.4.13.16.3.1. Definition: All DELETE tags

3.14.4.13.16.3.2. Action 1: If any DELETE TAQ matches, the HFS applies no change.

3.14.4.13.16.3.3. Action 2: If no DELETE TAQs match, multiply the DI_VAL by the TAQXSN Value.

Figure 39: Example: Multiple TAQ DELETES with Some Match

	REV DR VARGAS
REV VARGAS	1.00

Figure 40: Example: Multiple TAQ DELETEs with No Match

	GENERAL DR VARGAS
REV SR VARGAS	$1.00 * 0.85 = 0.85$

3.14.4.13.16.4. TAQ DISREGARDS

3.14.4.13.16.4.1. Definition: All DISREGARD tags

3.14.4.13.16.4.2. Action 1: If any TAQ DISREGARD segment matches, the HFS will make no change in the DI_VAL.

3.14.4.13.16.4.3. Action 2: If no TAQ DISREGARD segments match, the HFS will identify the highest match value from the TF (TF_VALUE) and multiply that by the DI_VAL.

Figure 41: Example: Multiple TAQ DISREGARDS with Matching TAQ Segment (DE LAS/DE LOS)

	DE LAS LUNAS
DE LOS LUNAS	1.00

Figure 42: Example: Multiple TAQ DISREGARDS with No Matching TAQ Segment (DE SANTA/LA)

	DE SANTA MARIA
LA MARIA	$1.00 * 0.75 = 0.75$

3.14.4.13.16.5. TAQ DISREGARD and DELETEs

3.14.4.13.16.5.1. Definition: Mixed DISREGARD/DELETE tags

3.14.4.13.16.5.2. Action 1: If DISREGARD segments are present in both comparands and there is any match among the DISREGARD segments, the HFS will make no change in the DI_VAL.

3.14.4.13.16.5.3. Action 2: If DISREGARD segments are present in both comparands and there is no match among the DISREGARD segments, the HFS will determine the highest match value from the TF for any DISREGARD tags and multiply the DI_VAL by that value. (That is, ignore any DELETE tags.)

3.14.4.13.16.5.4. Action 3: If a DISREGARD segment is in one comparand and not the other and the two comparands have at least one DELETE tag that matches, the HFS will make no change in the DI_VAL.

3.14.4.13.16.5.5. Action 4: If a DISREGARD segment is in one comparand and not the other and the two comparands have DELETE tags that do not match, multiply the DI_VAL by the TAQXSN.

Figure 43: Example: Multiple TAQs, DISREGARDs (DE/LOS)

	SR DE VARGAS
DR LOS VARGAS	$1.00 * 0.75 = 0.75$

Figure 44: Example: Multiple TAQs, DELETEs (DRA/DR)

	DRA DE VARGAS
DR VARGAS	$1.00 * 0.85 = 0.85$

3.14.4.14. After all parameters have been applied, the HFS will calculate the SN_VAL.

3.14.4.14.1. The HFS will choose the highest value for the row and column for any SN segments that have more than one digraph value assigned to them.

3.14.4.14.2. The HFS will sum the DI_VALs of all SN segments and will divide by the number of DI_VALs.

Figure 45: Example 1: Filter Evaluation

	GARCIA	GOMEZ
BOMEZ		$0.67 * 0.65 = 0.44$
GARCIA	$1.00 * 0.65 = 0.65$	

Figure 46: Example 2: Filter Evaluation

	GARCIA	GOMEZ
GAMEZ		$0.67 * 0.65 = 0.44$

Figure 47: Example 3: Filter Evaluation

	GARCIA	GOMEZ
GARZA	0.62	
GOMEZ		$1.00 * 0.65 = 0.65$

3.14.4.14.3. In Figure 45, $0.44 + 0.65 / 2 = 0.55$

3.14.4.14.4. In Figure 46, $0.44 / 1 = 0.44$

3.14.4.14.5. In Figure 47, $0.62 + 0.65 / 2 = 0.64$

3.14.4.15. The HFS will compare the SN_VAL to the SNTHR.

3.14.4.15.1. The SN_VAL must be equal to or greater than the SNTHR.

3.14.4.15.2. If the SNTHR were 0.60, only Example 3 above would pass.

3.14.4.15.3. The record must pass the SNTHR to qualify for Given Name Evaluation.

3.14.4.16. Given Name Evaluation

3.14.4.16.1. The HFS will evaluate the GN in a similar way to the SN evaluation.

3.14.4.16.2. The HFS will assign a DI_VAL of 1.00 to any match with FNU.

3.14.4.16.3. The GN format will permit more than two GN segments to be evaluated.

3.14.4.16.3.1. If the segment pair has a matching HFGN_KEY, the digraph value (DI_VAL) retrieved with that key will be assigned to the pair being evaluated.

3.14.4.16.3.2. For any segment pair that does not have a HFGN_KEY and associated DI_VAL, the DI_VAL will be calculated. (See Section 3.11.4.19.4 for digraph evaluation.)

3.14.4.16.3.3. The HFS will not calculate a digraph value for a GN_INIT Key value or GN initial.

3.14.4.16.3.3.1. The HFS will calculate the digraph relationship for all segments that have not been assigned a DI_VAL.

3.14.4.16.3.3.2. The HFS will not compare names that have a DI_VAL assigned.

Figure 48: Example: GN Digraph Evaluation

	MARIA	LORNA	SILVIA	CATERINA
CATHERINA				0.74 (HFGN_KEY)
MARIA	1.00 (HFGN_KEY)			
LARA		0.36	0.08	

MILDRED		0.00	0.07	
---------	--	------	------	--

3.14.4.16.4. The DI_VAL of each GN segment will be adjusted by several GN parameters.

3.14.4.16.5. **INITGN**: Given Name Initial

3.14.4.16.5.1. Definition 1: The GN segment is a single character in both comparands and the character matches exactly.

3.14.4.16.5.2. Action: The HFS will make no change.

3.14.4.16.5.3. Definition 2: A GN segment is a single character and its GN_INIT Key matches the GN_INIT Key of the other comparand.

3.14.4.16.5.4. Action: Assign the INITGN value to the comparison value (i.e., do not calculate the DI_VAL). The initial may be subjected to any following actions (e.g., out-of-place segment).

3.14.4.16.5.5. Definition 3: A GN segment is a single character and the GN_INIT Keys of the comparands do not match.

3.14.4.16.5.6. Action: Assign the INITNM value to the comparison value (i.e., do not calculate the DI_VAL). The initial may be subjected to any following actions (e.g., out-of-place segment).

3.14.4.16.6. **OPGVAL**: Out-of-Place Given Name Value

3.14.4.16.6.1. Definition: A GN segment that is not in the same relative position in the GN string in both the database and query records.

3.14.4.16.6.2. Action: Multiply the DI_VAL by the OPGVAL.

3.14.4.16.7. **AGVAL**: Anchor Given Name Value

3.14.4.16.7.1. Definition: For database records that contain two or more GN segments, the database SN segments are in the correct position relative to the query SN segments.

3.14.4.16.7.2. Action: Multiply the DI_VAL of the GN segments to the right of the first (leftmost segment) by the AGVAL.

Figure 49: Example 1: GN Parameter Evaluation: OPGN Applies

	MARIA	CATHERINA
KATHERINA		$0.90 * 0.65 = 0.59$
MARIA	$1.00 * 0.65 = 0.65$	

Figure 50: Example 2: GN Parameter Evaluation: OPGN Applies

	JOSE	BARTOLOMEO
BARTO		$0.71 * 0.65 = 0.46$

Figure 51: Example 3: GN Parameter Evaluation: AGVAL Applies

	JUAN	MARIO
JUANA	0.73	
MARIA		$0.83 * 0.65 = 0.54$

Figure 52: Example 4: Given Name Parameter Evaluation

	MARIA	LARA	MILDRED	CATERINA
CATHERINA				$0.74 * 0.65 = 0.48$ (OPGVAL)
MARIA	$1.00 * 0.65 = 0.65$ (OPGVAL)			
LORNA		$0.36 * 0.65 = 0.23$ (OPGVAL)	$0.08 * 0.65 = 0.05$ (OPGVAL)	
SILVIA		$0.00 * 0.65 = 0.00$ (OPGVAL)	$0.07 * 0.65 = 0.05$ (OPGVAL)	

3.14.4.16.8. TAQ Evaluation will proceed as with the SN, mutatis mutandi (See Section 3.14.4.13.7).

3.14.4.17. After all GN evaluations have been performed, the HFS will choose the highest score for each GN segment that has multiple DI_VALs (i.e., those for which no DI_VAL was retrieved with the key).

3.14.4.17.1. The highest score for both the row and column must be chosen.

3.14.4.17.2. Only one score per row and column is permitted.

3.14.4.17.3. If two scores are equal, only one is chosen.

3.14.4.17.4. In the example above, the higher score for LORNA is on the match with LARA (0.23); for SILVIA, MILDRED (0.05).

3.14.4.17.4.1. Note that MILDRED scores are equal, but the row for LORNA has already been chosen.

3.14.4.17.4.2. Only one value can be chosen for each row and column.

3.14.4.18. The HFS will sum all DI_VALs from the comparison matrix and will divide by the number of DI_VALs to produce the GN score.

3.14.4.18.1. In Example 1, $0.59 + 0.65/2 = 0.62$

3.14.4.18.2. In Example 2, $0.46/1 = 0.46$

3.14.4.18.3. In Example 3, $0.73 + 0.54/2 = 0.64$

3.14.4.18.4. In Example 4, $0.48 + 0.65 + 0.23 + 0.05/4 = 0.33$

3.14.4.19. The HFS will further evaluate the Given Name by comparing the record gender of the two comparands.

3.14.4.19.1. If the record genders match, no action will take place.

3.14.4.19.2. If the record genders do not match, the HFS will apply the RGNDR value to the GN score.

3.14.4.19.2.1. The HFS will access the TF to determine the RGNDR Value.

3.14.4.19.2.2. The HFS will multiply the GN score by the RGNDR value.

3.14.4.20. The value resulting from the full GN evaluation will be the GN_VAL.

3.14.4.21. The HFS will compare the GN_VAL to the GNTHR.

3.14.4.21.1. The GN_VAL must be equal to or greater than the GNTHR.

3.14.4.21.2. The GN_VAL must pass the GNTHR for the record to qualify for calculation of the Composite Score.

3.14.4.22. Composite Score

3.14.4.23. The HFS will develop a composite score for two comparands that will reflect the proximity of the query and database *records*.

3.14.4.24. The Composite Score will be used to rank order the records being evaluated.

3.14.4.25. The Name is one component of the Composite Score; others are the Refusal Level, Date-of-Birth and Country of Birth.

3.14.4.25.1. The HFS will adjust the GN_VAL and the SN_VAL by factors that reflect the proximity of the Refusal Level, Date-of-Birth and Country of Birth.

3.14.4.25.2. The GN_VAL and SN_VAL will be multiplied by RL, YOB and COB factors.

3.14.4.26. Refusal Level Factor

3.14.4.27. The HFS will access the Refusal Code Level Data Store to determine the Refusal Level Category of the Refusal Code.

3.14.4.28. The HFS will access the Hispanic Parameter Data Store to find the PARM_VAL associated with the Refusal Level (RL#).

3.14.4.29. Date-of-Birth Factor

3.14.4.30. The HFS will access the Year-of-Birth Range Data Store to determine the YOB Category, YOB#, of the Dates-of-Birth of the comparands. The highest value is applied to the relationship.

3.14.4.31. The HFS will access the Hispanic Parameter Data Store to find the PARM_VAL associated with the YOB Category (YOB#).

3.14.4.32. Country-of-Birth Factor

3.14.4.33. The HFS will access the Hispanic Country-of-Birth Category Data Store (HCOB) to determine the COB Category, COB#.

3.14.4.33.1. The HFS will identify the COB#.

3.14.4.33.2. The HFS will access the Hispanic Parameter Data Store to find the PARM_VAL associated with the Country-of-Birth Category (COB#).

3.14.4.34. Calculating the Composite Score

3.14.4.35. The HFS will calculate a Composite Score by multiplying the SN_VAL by the GN_VAL by the RL# PARM_VAL by the YOB# PARM_VAL by the COB# PARM_VAL.

3.14.4.36. Final Sort Function of the HFS

3.14.4.37. The HFS will order the final candidate list.

3.14.4.38. The HFS will place at the top of the candidate list all records that have been tagged as exact matches.

3.14.4.39. The HFS will then rank order in descending order of Composite Score all records for which a Composite Score has been calculated.

3.14.4.39.1. The goal of the final sort is to place exact record matches on the top and to rank order the remaining records by the degree of contribution that each data element (SN, GN, DOB, COB, Refusal Code Level (RL)) makes to the overall record value.

3.14.4.39.2. Further details of the sort will be derived from extensive discussion about the business requirements.

3.14.4.39.3. Because the scores from the various pipes may not have been calculated in the same way, a method for evaluating the relative value of candidate records will have to be devised.

3.14.4.40. Internal Sort Order

3.14.4.40.1. There may be cases in which the sorting criteria are met equally by more than 1 record.

3.14.4.40.2. Where multiple records qualify equally, there will be an internal sort order.

3.14.4.40.2.1. SN Score

3.14.4.40.2.2. GN Score

3.14.4.40.2.3. DOB Level

3.14.4.40.2.4. Refusal Code Level

3.14.4.40.2.5. COB Relationship

3.14.4.41. The HFS will return the top n records to the central CLASS-E sorter.

3.14.4.41.1. The number of records to be returned will be a system setting.

3.15. LINGUISTIC TRACE FACILITY MODULE DECOMPOSITION

3.15.1. Identification

This module is known as the Linguistic Trace Facility (LTF).

3.15.2. Type

The LTF is a program that will interact with any or all modules and functions within those modules.

3.15.3. Purpose

The LTF will allow system evaluators to access information about the system functions so that the quality of the content can be ensured. To diagnose and remedy problems associated with questionable system results, evaluators must have access to the results of system functionality at various points during the processing cycle.

3.15.4. Function

3.15.4.1. The LTF will be a mechanism that will copy and divert statistics, information, processing results to a file outside the main processing module.

3.15.4.2. The file will be readily accessible for on-line examination by system evaluators.

3.15.4.3. Multiple trace points will be identified when the system is built.

3.15.4.4. Examples of trace points:

- Derived record formats
- All keys generated for a query and for an add

- Records qualifying with the LFDI_KEY
- SN and GN DI_VAL
- SN_VAL and GN_VAL
- Record Gender
- RL#, YOB#, COB# Values
- Sort considerations.

4. DATA DECOMPOSITION

4.1. DATA

4.1.1. The input data for an HNA-E query will contain all information that is currently required by CLASS.

4.1.2. The input data for an HNA-E query will be in the standard format currently required by CLASS.

- NAME (Surname, Given Name);
 - The SN is a required name field and therefore must be filled.
 - Last Name Unknown (LNU) is not a permitted string in HNA-E.
 - The SN may be represented by a single character, which will be interpreted as an initial.
- DOB (Date of Birth; Day Month Year); and
- COB (Country of Birth; FIPS codes).

4.1.3. In addition, the following will be specified:

- Applicant Gender (AG): Male (M), Female (F), Unknown/Ambiguous (U).
- A unique identifier (UID) (as defined in CLASS-E).

4.1.4. For record adds, additional record information will be entered, as required by CLASS and CLASS-E: e.g., refusal code, province of birth.

4.2. DATA COLLECTION

4.2.1. Two alternative approaches to tagging the name data are available: the name as an object and the name as a data element.

4.2.2. The system could define the name as an object that knows something about itself and collects information as it passes through the various processing modules.

4.2.2.1. A name object would make the relevant information available to the various processing modules, as needed, from one consistent, predefined object.

4.2.2.2. A name object may also permit the same name to be handled in the same way on another occasion. Reuse of information would be especially valuable for HF names.

4.2.3. The second, alternative approach is to tag the specific items as they undergo processing or change, to access information in data stores as it is needed, and to tag the name or name segment for the relevant processes it undergoes.

4.3. DATA STORES

4.3.1. Several of the Data Stores proposed could be collapsed into one data store (e.g., the HF SN Data Stores: HFST and HFSV); for ease and clarity of exposition and reference, the data stores have been maintained separately.

4.3.2. HNA-E will access X Data Stores:

- Hispanic TAQ Data Store (HTD)
- High Frequency Surname Type Data Store (HFST)
- High Frequency Surname Variant Data Store (HFSV)
- Low Frequency Surname Type Data Store (LFST)
- Hispanic Given Name Type Data Store (HGT)
- High Frequency Given Name Variant Data Store (HFGV)
- Hispanic Character Data Store (HCD)
- Hispanic Parameter Data Store (HPD)
- High Frequency Decision Matrix (HDM)
- Refusal Code Level Data Store (RCL)
- Year-of-Birth Range Data Store (YR)
- Refusal Code Level/Year-of-Birth Range Data Store (RLYOB)
- Country-of-Birth Proximity Data Store (COBPROX)
- Hispanic Country-of-Birth Category Data Store (HCOB)

4.4. HISPANIC TITLE/AFFIX/QUALIFIER DATA STORE DECOMPOSITION

Because the HNA-E design is viewed as an independent sub-program of the CLASS-E system, the Hispanic Title/Affix/Qualifier Data Store is presented here as

a separate table. It is strongly suggested, however, that CLASS-E support one TAQ Data Store in which the cultural affinity of each TAQ segment is indicated. This will reduce table maintenance and will provide a global picture of the handling of TAQs.

4.4.1. Identification

This data store is known as the Hispanic Title/Affix/Qualifier (TAQ) Data Store (HTD).

4.4.2. Type

4.4.2.1. The HTD is a data store that contains the Hispanic-specific Title, Affix and Qualifier segments with additional information about the disposition of the items.

4.4.2.2. The HTD will be accessed by the Hispanic Name Preprocessor (HNP) and the Hispanic Filter and Sorter (HFS).

4.4.2.3. The format of the HTD will be

Figure 53: Format: Hispanic TAQ Data Store (HTD)

DATA FIELD	DATA TYPE	FIELD SIZE	DATA RANGE/VALUE
ID_NO	integer	4	0...9999
TAQ_FORM	character	15	alphabetic
TAQ_TYPE	character	1	T, I, P, S, Q
DELETE	integer	1	1, 0 (True, False)
DISREGARD	integer	1	1, 0 (True, False)
REMOVE	integer	1	1, 0 (True, False)

4.4.2.4. Definitions

4.4.2.4.1. ID_NO: a unique, arbitrary number that identifies the TAQ segment.

4.4.2.4.2. TAQ_FORM: the string that represents the TAQ; the TAQ_FORM may be a multipart string (i.e., a string that includes internal white space).

4.4.2.4.3. TAQ_TYPE: an indicator of the kind of TAQ segment present: a title (T), prefix (P), infix (I), suffix (S), or qualifier (Q).

4.4.2.4.4. DELETE:

4.4.2.4.4.1. The segment is to be removed from all further consideration in the name search process.

4.4.2.4.4.2. The segment is referenced in the filtering process.

4.4.2.4.4.3. The segment is not removed from the original record and is returned with the record to the user.

4.4.2.4.4. True (1) or False (0) indicates whether or not this function is to apply to the segment(s) under consideration.

4.4.2.4.5. DISREGARD:

4.4.2.4.5.1. The segment is to be removed from further consideration in the name search process but will undergo special evaluation in the filtering process. It will be returned with the record to the user.

4.4.2.4.5.2. True (1) or False (0) indicates whether or not this function is to apply to the segment(s) under consideration.

4.4.2.4.6. REMOVE:

4.4.2.4.6.1. The segment occurs attached to the name stem.

4.4.2.4.6.2. The conjoined TAQ will be separated from a base name segment. (See Section 3.5.4.4.3).

4.4.2.4.6.3. True (1) or False (0) indicates whether or not this function is to apply to the segment(s) under consideration.

4.4.2.4.6.4. The separated segment will also be marked for DELETE/DISREGARD treatment.

4.4.3. Purpose

Peripheral elements (Titles, Affixes, and Qualifiers) in names do not contribute as much to the name evaluation as does the name stem. Identifying and removing these elements in the name processing component is important. They do, however, contribute to the overall value of a name when determining the proximity of one name to another. They will therefore contribute some value to the filtering and sorting processes.

4.4.4. Function

The HTD serves as a repository for all TAQ values and for the treatment that each will be subjected to.

4.5. HIGH FREQUENCY SURNAME TYPE DATA STORE DECOMPOSITION

4.5.1. Identification

4.5.1.1. This data store is known as the High Frequency Surname Type Data Store (HFST).

4.5.1.2. This data store could be merged with the High Frequency Surname Variant Data Store (HFSV).

4.5.1.2.1. The ID_NO would be different in the HFSV and would serve as a unique identifier for each entry.

4.5.1.2.2. The set of HFSN_TYPES, with no variants, would be derivable from the HFSN_VARS with a DI_VAL equal to 1.00.

4.5.1.2.3. The SET_ID of the HFST and HFSV would be the same.

4.5.2. Type

4.5.2.1. The HFST data store consists of the 500 most frequently occurring HF SN segment types (i.e., unique occurrence).

4.5.2.2. The HFST will be accessed by the Hispanic Surname Segmenter, Hispanic Segment Positioner, and the Frequency Path Director modules.

4.5.2.3. The format of the HFST will be

Figure 54: Format: High Frequency Surname Type Data Store (HFST)

DATA FIELD	DATA TYPE	FIELD SIZE	VALUE RANGE
ID_NO	integer	4	1...9999
HFSN_TYPE	character	24	alphabetic
SET_ID	integer	4	1...9999

Figure 55: Example: Piece of HFST

ID_NO	HFSN_TYPE	SET_ID
0001	GARCIA	0001
0002	RODRIGUEZ	0002
0003	HERNANDEZ	0003
0004	LOPEZ	0004
0005	MARTINEZ	0005
0006	GONZALEZ	0006
0007	PEREZ	0007
0008	SANCHEZ	0008
0009	RAMIREZ	0009
0010	GOMEZ	0010
0011	...	0011

4.5.2.4. Definitions

4.5.2.4.1. ID_NO will be a unique numerical identifier for each of the HF SN segments, HFSN_TYPES.

4.5.2.4.2. HFSN_TYPE will contain a unique character string that represents one of the 500 most frequently occurring Hispanic surname stems.

4.5.2.4.3. SET_ID will be the unique identifier for the set of variants of the HFSN_TYPE and will be used as the HFSN_KEY.

4.6. HIGH FREQUENCY SURNAME VARIANT DATA STORE DECOMPOSITION

4.6.1. Identification

4.6.1.1. This data store is known as the High Frequency Surname Variant Data Store (HFSV).

4.6.1.2. This data store will be updated in real time as variants qualify for inclusion in the data store. (See Section 3.12.4.38)

4.6.1.3. This data store could be merged with the HFST (See Section 4.5.1.2.)

4.6.2. Type

4.6.2.1. The HFSV is a data store that consists of a HFSN_TYPE segment with a variant of that segment and a value that represents the degree of digraph proximity of the HFSN_TYPE and its variant.

4.6.2.2. The HFSV is a data store that will have between 75,000 and 100,000 rows.

4.6.2.3. A name segment may be the variant of more than one HFSN_TYPE.

4.6.2.4. The HFSV will be accessed by the High Frequency Processor and Low Frequency Processor.

4.6.2.5. The format of the HFSV will be

Figure 56: Format: High Frequency Surname Variant Data Store (HFSV)

DATA FIELD	DATA TYPE	FIELD SIZE	VALUE RANGE
ID_NO	integer	6	000000...999999
HFSN_VAR	character	24	alphabetics
SET_ID	integer	4	0000...9999
DI_VAL	decimal	4	0.00...1.00

Figure 57: Example: High Frequency Surname Variant Data Store

ID_NO	HFSN_VAR	SET_ID	DI_VAL
032711	PEREZ	0007	1.00
032712	PERES	0007	0.67
032713	PEREZA	0007	0.77
016976	GOMEZ	0010	1.00
016977	GOMES	0010	0.67
016978	BOMEZ	0010	0.67

4.6.2.6. Definitions

4.6.2.6.1. ID_NO will be a unique numerical identifier for each HFSN_VAR entry.

4.6.2.6.2. HFSN_VAR will contain a character string that has been determined to be a variant of the HFSN_TYPE with which it is associated. A HFSN_VAR may be a variant of one or more of the HFSN_TYPES.

4.6.2.6.2.1. A variant is defined as a name stem that shares a sufficient number of digraphs (strings of two characters) with the HFSN_TYPE to pass a pre-determined threshold.

4.6.2.6.2.2. A HFSN_TYPE can be obtained from the HFST or from the HFSV as a HFSN_VAR with a DI_VAL equal to 1.00.

4.6.2.6.3. DI_VAL is a two-place decimal value that represents the proximity of the HFSN_TYPE and the HFSN_VAR.

4.6.2.6.3.1. The DI_VAL is a calculation derived from the shared digraphs (strings of two characters) of the HFSN_TYPE and the HFSN_VAR associated with it.

4.6.2.6.3.2. The calculation is determined in the following way:

4.6.2.6.3.2.1. The digraphs are identified for each name stem, the HFSN_TYPE (Comparand #1) and the HFSN_VAR (Comparand #2).

4.6.2.6.3.2.1.1. Each pair of alphabetic characters is identified: GOMEZ
→ GO / OM / ME / EZ

4.6.2.6.3.2.1.2. A digraph is also formed of the initial boundary (#) and the first alphabetic character:
GOMEZ → #G.

4.6.2.6.3.2.1.3. A digraph is also formed of the final alphabetic character and the final boundary (#): GOMEZ
→ Z#.

4.6.2.6.3.2.2. The number of shared digraphs is calculated; a digraph may match one digraph only.

4.6.2.6.3.2.3. The number of shared digraphs is multiplied by 2 and divided by the total number of digraphs in comparand #1 added to the total number of digraphs in comparand #2.

4.6.2.6.3.2.4. The formula is:

$2 * d / a + b$, where d = the total number of shared digraphs; where a = the total number of digraphs in Comparand #1 and where b = the total number of digraphs in Comparand #2.

Figure 58: Example: Digraph Evaluation of Two Comparands

COMPARANDS	DIGRAPHS	SHARED DIGRAPHS (d)	DI_VAL
COMPARAND #1: DOMINGUEZ	#D DO OM MI IN NG GU UE EZ Z#	#D DO OM MI IN UE	$2 * d / a + b = 12 / 20$
COMPARAND #2: DOMINQUES	#D DO OM MI IN NQ QU UE ES S#	= 6	.60

4.6.3. Purpose

4.6.3.1. For HNA-E to be an effective retrieval system, it must be able to retrieve variants of query names. The impact on system performance can be dramatic, however, if traditional matching techniques are used to identify variant names. By assigning variants to the same set and recording their digraph value, querying a HF surname will result in the direct retrieval of variant records and their digraph values.

4.6.3.2. The HFSV also serves as a resource for identifying which HF surnames are related to a LF surname.

4.6.4. Function

The HFSV Data Store will be dynamically updated. (See Section 3.12.4.38 for details.)

4.7. HIGH FREQUENCY DECISION MATRIX DATA STORE

4.7.1. Identification

This data store is known as the High Frequency Decision Matrix (HDM).

4.7.2. Type

4.7.2.1. The HDM is a data store that will provide criteria for database record retrieval for query records with HF name segments.

4.7.2.2. It will be accessed by the High Frequency Processor (HFP).

Figure 59: Format: Hispanic Decision Matrix (HDM)

DATA FIELD	DATA TYPE	FIELD SIZE	VALUE RANGE
QUERY SN FORMAT	character	2	A, B
DATABASE SN FORMAT	character	2	A, B, C
YOB RANGE (YOB#)	integer	1	0...6
REFUSAL CODE LEVEL (RL#)	integer	1	0...4
RECORD GENDER (RGNDR)	character	3	M, F, U

Figure 60: Example: Hispanic Decision Matrix (HDM) (Values for example only)

	Single-Segment SN			Two-Segment SN							
	A	A	A	AB	AB	AB	AB	AB	AB	AB	AB
QUERY SN FORMAT	A	AB	BA	AB	BA	A	B	AC	CA	CB	BC
DATABASE SN FORMATS	5	5	2	5	4	4	2	2	0	0	0
YOB#	4	4	3	4	4	4	1	1	0	0	0
RL#	MFU	MFU	MFU	MFU	MFU	MFU	MFU	FU	MFU	MFU	MFU
RGNDR											

4.7.3. Definitions

4.7.3.1. QUERY SN FORMAT: is a character string that is an abstract representation of the query SN. Each segment is represented by a single character, the leftmost A, the next B. The sequence also represents the position of the segment.

4.7.3.2. DATABASE SN FORMAT: is a character string that is an abstract representation of the possible and acceptable variations in the query SN which are relevant to the QUERY SN FORMAT and which will be retrieved from the database, given the conditions stipulated in the YOB RANGE (YOB#), REFUSAL CODE LEVEL (RL#) and RECORD GENDER (RGNDR). Each segment is represented by a single character.

4.7.3.2.1. If the character is the same as a character in the QUERY SN FORMAT, it represents the same SN Key.

4.7.3.2.2. If the character is different from a character in the QUERY SN FORMAT, it represents a different SN Key.

4.7.3.2.3. If the character is in the same relative position as that in the query SN, it represents the same position in the SN string.

4.7.3.2.4. If the character is not in the same relative position as that in the query SN, it represents a different (out-of) position in the SN string.

4.7.3.3. YOB RANGE (YOB#): is an integer that represents a YOB range specified in the YOB RANGE (YR) Data Store. (N.B. In this scheme, YOB# integer does not represent the year range itself. It refers to a table that specifies that YOB2, for example, represents an exact year-of-birth and that YOB3 represents a range of 1 year on either side of the query year (for a range total of 3 years).)

4.7.3.4. REFUSAL CODE LEVEL (RL#): is an integer that represents a Refusal Code Level specified in the REFUSAL CODE LEVEL Data Store. (N.B. In this scheme, this number represents a set of Refusal Codes that has a pre-determined degree of seriousness. The number given here does not signal the Refusal Code itself. The number is expanded in the Refusal Code Level Data Store, where 0, for example, might represent a 00 Refusal Code.)

4.7.3.5. RECORD GENDER (RGNDR): is a set of up to three characters that represent the required Record Gender of the database record.

4.7.4. Purpose

Many Hispanic surnames occur with very high frequency; they also generally have at least two segments. Any retrieval system that captures only one of these names will have an inordinately high recall. Many of these records will not be at all relevant to the query record. Special treatment of high frequency names must entail some method of reducing the number of irrelevant records retrieved from the database. The HDM provides the information about how to delimit the records that will be retrieved from the database. A reduction in the recall will reduce post-processing time.

4.7.5. Function

The HDM is a data store that consists of qualifying and delimiting criteria.

4.7.5.1. Qualifying criteria will be the number of SN segments, SN content, and SN segment positions.

4.7.5.2. Delimiting criteria will be Year-of-Birth (YOB) Range (YR), Refusal Code (RC) Level (RL) and Record Gender (RGNDR).

4.7.5.2.1. The qualifying criteria will produce a set of SN formats to retrieve from the database.

4.7.5.2.2. The delimiting criteria will specify the YOB range, maximum RC Level for each of the SN formats and Record Gender limitations, if any.

4.8. HISPANIC GIVEN NAME TYPE DATA STORE DECOMPOSITION

4.8.1. Identification

4.8.1.1. This data store is known as the Hispanic Given Name Type Data Store (HGT).

4.8.1.2. This data store could be merged with the High Frequency Given Name Variant Data Store (HFGV).

4.8.1.2.1. The ID_NO would be different in the HFGV and would serve as a unique identifier for each entry.

4.8.1.2.2. The set of HFGN_TYPERs, with no variants, would be derivable from the HFGN_VARs with a DI_VAL equal to 1.00.

4.8.1.2.3. The SET_ID of the HFGT and HFGV would be the same.

4.8.2. Type

4.8.2.1. The HGT data store will consist of up to ten thousand entries.

4.8.2.2. The HGT will be accessed by the Hispanic Gender Identifier, Frequency Path Director (FPD), the High Frequency Processor.

4.8.2.3. The HGT will have the following format:

Figure 61: Format: Hispanic Given Name Type Data Store (HGT)

DATA FIELD	DATA TYPE	FIELD SIZE	VALUE RANGE
ID_NO	integer	4	0000...9999
GN_TYPE	character	24	alphabets
SET_ID	integer	3	001...999
HI_FREQ	integer	1	1, 0 (True, False)
GNDR	character	1	M, F, U

Figure 62: Example: Hispanic Given Name Type Data Store (HGT)

ID_NO	GN_TYPE	SET_ID	HI_FREQ	GNDR
0001	JOSE	0001	1	M
0002	MARIA	0002	1	F
0003	JUAN	0003	1	M
0004	LUIS	0004	1	M
0005	ANTONIO	0005	1	M
0006	CARLOS	0006	1	M
0007	JESUS	0007	1	M
0008	MANUEL	0008	1	M
0009	FRANCISCO	0009	1	M
0010	JORGE	0010	1	M
0011	...	0011		...
2367	DAGOBERTO	0000	0	M

4.8.2.4. Definitions

4.8.2.4.1. ID_NO: is an integer that is a unique numerical identifier for each of the GN_TYPES.

4.8.2.4.2. GN_TYPE: is a character string that represents one of up to ten thousand Hispanic given name stems.

4.8.2.4.2.1. A HFGN_TYPE is a GN_TYPE whose HI_FREQ value is 1 (True).

4.8.2.4.3. SET_ID: is an integer that is the numerical identifier for the set of related variants of the GN_TYPE that is HF.

4.8.2.4.3.1. The SET_ID will serve as the HFGN_KEY.

4.8.2.4.3.2. Not every entry in the HGT will have a unique SET_ID; a distinct SET_ID is reserved for those GN_TYPES where HI_FREQ is True (1).

4.8.2.4.4. HI_FREQ: is an integer (1, 0/True, False) that indicates if the GN_TYPE is or is not a HF GN segment.

4.8.2.4.4.1. The frequency of all GN_TYPES will be specified.

4.8.2.4.4.2. True (1) will indicate a HF segment.

4.8.2.4.4.3. False (0) will indicate a LF segment.

4.8.2.4.5. GNDR: is a single character value that indicates the gender of the GN_TYPE.

4.8.2.4.5.1. If the name is predictably female, the value will be F.

4.8.2.4.5.2. If the name is predictably male, the value will be M.

4.8.2.4.5.3. If the name is ambiguous or unknown, the value will be U.

4.8.3. Purpose

The HGT provides information about Hispanic given name segments. It indicates the frequency of the segments, their gender and the set of names of which they are the parent.

4.8.4. Function

The HGT serves as a resource for Hispanic Gender Identifier and the High Frequency Processor.

4.9. HIGH FREQUENCY GIVEN NAME VARIANT DATA STORE DECOMPOSITION

4.9.1. Identification

This data store is known as the High Frequency Given Name Variant Data Store (HFGV).

4.9.2. Type

4.9.2.1. The HFGV will be accessed by the High Frequency Processor.

4.9.2.2. The HFGV will have about 60,000 to 90,000 rows.

4.9.2.3. The HFGV will have the following format:

Figure 63: Format: High Frequency Given Name Variant Data Store (HFGV)

DATA FIELD	DATA TYPE	FIELD SIZE	VALUE RANGE
ID_NO	integer	5	00000...99999
HFGN_VAR	character	24	alphabetics
SET_ID	integer	4	0000...9999
DI_VAL	decimal	4	0.00...1.00

Figure 64: Example: Piece of HFGV

ID_NO	HFGN_VAR	SET_ID	DI_VAL
00001	JOSE	0001	1.00
00002	JOSEA	0001	0.73
00003	JOSSE	0001	0.73
00004	MARIA	0002	1.00
00005	MIRIA	0002	0.67
00006	MIRIAM	0164	1.00
00007	MIRIA	0164	0.77

4.9.2.4. Definitions

4.9.2.4.1. ID_NO: is a unique numerical identifier for each HFGN_VAR entry in the HFGV data store.

4.9.2.4.2. HFGN_VAR: is character string that represents a GN segment that is a digraph variant of the HFGN_TYPE (HFGN_VAR whose DI_VAL = 1.00).

4.9.2.4.3. SET_ID: is a unique identifier of the set of GN segments that are variants of the same HFGN_TYPE.

4.9.2.4.4. DI_VAL: is a two-place decimal value that indicates the digraph relationship between the HFGN_VAR and its parents HFGN_TYPE.

4.9.3. Purpose

The HFGV is a resource for defining the given name segments that will be stored with records added to the database. Storage of information about variant relations will speed retrieval and the filtering process.

4.9.4. Function

The HFGV will be accessed by the HFP to assign keys to given name segments on record add and query.

4.10. LOW FREQUENCY SURNAME TYPE DATA STORE DECOMPOSITION

4.10.1. Identification

This data store is known as the Low Frequency Surname Type Data Store (LFST).

4.10.2. Type

4.10.2.1. The LFST is a data store of LF keys.

4.10.2.2. The LFST will have about 900,000 to 1 million rows.

4.10.2.3. The LFST will have the following format:

Figure 65: Format: Low Frequency Surname Type Data Store (LFST)

DATA FIELD	DATA TYPE	FIELD SIZE	VALUE RANGE
ID_NO	integer	6	000001...999999
LFSN_TYPE	character	24	alphanumerics
LFDIKEY	character	3	alphanumerics; char + char + #

Figure 66: Example: Piece of LFST

ID_NO	LFSN_TYPE	LFDIKEY
000001	AALVAREZ	AA1
000001	AALVAREZ	AA2
000001	AALVAREZ	AL2
000001	AALVAREZ	AL1
000001	AALVAREZ	AL3
000001	AALVAREZ	LV3
000001	AALVAREZ	LV2
000001	AALVAREZ	LV4
000001	AALVAREZ	VA4
000001	AALVAREZ	VA3
000098	BARRIOS	BA1
000098	BARRIOS	BA2
...		

4.10.2.4. Definitions:

4.10.2.4.1. ID_NO: is an arbitrary numerical reference to each LFSN_TYPE. The ID_NO will serve as the DI_KEY.

4.10.2.4.2. LFSN_TYPE: is the unique low frequency name segment as it occurs in the database; if there are multiple occurrences of the same name, they are represented by one entry, hence the term "type."

4.10.2.4.3. LFDIKEY: is a string of alphanumeric characters that represents one digraph and its actual or derived position.

4.10.2.4.3.1. Up to ten LFDIKEYs will be associated with each LFSN_TYPE.

4.10.2.4.3.2. An LFDIKEY is name-specific, so the same key may appear with other LFSN_TYPES, in which case it will have a different ID_NO.

4.10.2.4.3.3. A LFDIKEY is

- 1) a digraph formed from the LF SN segment beginning with the leftmost character and its position (Base Key) and
- 2) a positional variant on that digraph key (Position Key).

4.10.2.4.3.4. Positional information will be associated with each digraph.

4.10.2.4.3.5. To form a key, begin with the leftmost character and generate four digraph keys (Base Key) from the five leftmost characters of the LF SN segment. The first two characters form a digraph, the second and third characters form a digraph, the third and fourth characters form a digraph and the fourth and fifth characters form a digraph. Positional information (Positions 1, 2, 3, 4) will be included.

4.10.2.4.3.6. Generate, from the Base Keys, up to six additional Position Keys; the position keys have the same characters as the Base Keys but contain different positional information. A maximum of ten keys (Base + Position) will be generated.

4.10.2.4.3.6.1. Produce a Position Key on the first Base Key with Position 2.

4.10.2.4.3.6.2. Produce Position Keys on the second Base Key with Position 1 and Position 3.

4.10.2.4.3.6.3. Produce Position Keys on the third Base Key with Position 2 and Position 4.

4.10.2.4.3.6.4. Produce a Position Key on the fourth Base Key with Position 3. No Position Key is generated for Position 5 because the maximum of 10 keys has been reached.

4.10.3. Purpose

The LFST provides information that will limit the search of database records. Preprocessing of name types allows identification of relevant name segments without having to examine database records directly.

4.10.4. Function

The LFST will be accessed by the LFP.

4.11. HISPANIC CHARACTER DATA STORE

4.11.1. Identification

This data store is known as the Hispanic Character Data Store (HCD).

4.11.2. Type

4.11.2.1. The HCD is a data store of all characters in Hispanic names and their predictable variants.

4.11.2.2. The format of the HCD will be:

Figure 67: Format: Hispanic Character Data Store (HCD)

DATA FIELD	DATA TYPE	FIELD SIZE	VALUE RANGE
SET_ID	integer	3	000...999
CHAR	character	1	alphabetics
CHAR_VAR	character	1	alphabetics

Figure 68: Example: Piece of HCD

SET_ID	CHAR	CHAR_VAR
001	B	B
001	B	V
002	S	S
002	S	Z
004	C	C
004	C	S
...		
037	F	F
052	K	K
078	M	M
078	M	N
...		

4.11.2.3. Definitions

4.11.2.3.1. SET_ID: is an arbitrary numerical that represents the set of characters that vary with one another. The SET_ID will be the GN_INIT Key.

4.11.2.3.2. CHAR: is a single alphabetic character. Every alphabetic character will be represented. The CHAR is the type of

character, which may or may not have variants (CHAR_VAR).

4.11.2.3.3. CHAR_VAR: is a single alphabetic character that may or may not vary predictably with other characters in written Spanish. A single character may participate in more than one set.

4.11.3. Purpose

Retrieval of records with HF SN segments from the database will be limited by the initial of the GN segments. For the retrieval to be sufficiently robust, however, the system must allow for some variation in the GN initials. The HCD indicates variations on initials.

4.11.4. Function

The HCD will be accessed by the HFP and will provide the source of the GN_INIT Keys that are to be generated for HF searches.

4.12. TAQ FILTER DATA STORE DECOMPOSITION

4.12.1. Identification

This data store is known as the TAQ Filter Data Store (TF).

4.12.2. Type

4.12.2.1. This TF will be accessed by the Hispanic Filter and Sorter and provides parameter factors for matching TAQ DISREGARD tags during record filtering.

4.12.2.2. The format of the TF follows:

Figure 69: Format: TAQ Filter Matrix Design

DATA FIELD	DATA TYPE	FIELD SIZE	VALUE RANGE	DATA VALUE
TAQDIS#1	character	8	alphabetics	TAQ_DISREGARD ITEM
TAQDIS#2	character	8	alphabetics	TAQ_DISREGARD ITEM
TF_VALUE	decimal	4	0.00...1.00	Various (TBD)

Figure 70: Example: Piece of TF (Values are for example only)

TAQDIS#1	TAQDIS#2	TF_VALUE
DE	DE	1.00

DE	DEL	0.90
DE	DE LOS	0.90
DE	LOS	0.75
DE	SAN	0.75
DE	LA	0.75
DEL	DEL	1.00
DEL	DE LOS	0.75
DEL	LOS	0.65
DEL	LA	0.85
DEL	SAN	0.50
DE LOS	DE LOS	1.00
DE LOS	LOS	0.90
DE LOS	SAN	0.50
DE LOS	LA	0.50
SAN	SAN	1.00
SAN	LOS	0.50
SAN	LA	0.50
LOS	LOS	1.00
LOS	LA	0.85
LA	LA	1.00
...		

4.12.2.3. Definitions

4.12.2.3.1. TAQDIS#1: is the TAQ DISREGARD segment that occurs in one or the other (different) of the comparands.

4.12.2.3.2. TAQDIS#2: is the TAQ DISREGARD segment that occurs in one or the other (different) of the comparands.

4.12.2.3.3. TF_VALUE: is the factor that will be used to adjust the SN_VAL or GN_VAL if the TAQDIS#1 and TAQDIS#2 are present in the comparands.

4.12.3. Purpose

Hispanic names often have peripheral name elements. Some of these make up a segment of the name, the TAQ values identified in the TF. Their relative value, however, varies. Some of them cannot cooccur, some have opposite meanings, so it is necessary to identify their relative value when they are contrasted with one another.

4.12.4. Function

The TF provides the resources for the HFS to determine the relative value of TAQs that occur in two comparands.

4.13. HISPANIC PARAMETER DATA STORE DECOMPOSITION

4.13.1. Identification

This module is known as the Hispanic Parameter Data Store (HPD).

4.13.2. Type

4.13.2.1. The HPD is a data store that will be accessed by the Filter Component of the Hispanic Filter and Sorter (HFS).

4.13.2.2. The HPD is a parameter table that will be accessible to the user and whose cell values will be determined through testing and comparative evaluation.

4.13.2.3. The HPD has the following format:

Figure 71: Format: Hispanic Parameter Data Store (HPD)

DATA FIELD	DATA TYPE	FIELD SIZE	VALUE RANGE	DATA VALUE
PARM_NAME	character	6	alphabetics	SNTHR, GNTHR, OPSVAL, OPGVAL, INITSN, INITGN, RGNDR, TAQASN, TAQAGN, TAQXSN, TAQXGN, RL#, YOB#, COB#, etc.
PARM_VAL	decimal	4	0.00...1.99	Various (TBD)

Figure 72: Example: HPD (Values are for example only.)

PARM_NAME	PARM_VAL
SNTHR	0.60
GNTHR	0.65
LFDIKEY THRESHOLD	0.57

DI_VAL THRESHOLD	0.63
HFGV THRESHOLD	0.65
HFSV THRESHOLD	0.65
OPGVAL	0.60
OPSVAL	0.60
ASVAL	0.65
AGVAL	0.65
INITSN	0.85
INITGN	0.85
INITNM	0.80
RGNDR	0.65
TAQASN	0.90
TAQAGN	0.90
TAQXSN	0.85
TAQXGN	0.85
RL0	1.20
RL1	1.15
RL2	1.10
RL3	1.05
RL4	1.00
YOB0	1.30
YOB1	1.25
YOB2	1.20
YOB3	1.15
YOB4	1.10
YOB5	1.05
YOB6	1.00
COB1	1.20
COB2	1.15
COB3	1.10
COB4	1.00
COB5	0.95

4.13.2.4. The values provided are for example only and do not necessarily represent the PARM_VALs to be used for the parameters.

4.13.3. Purpose

The HPD is a data store that allows easy access to adjustable thresholds for record qualification, to thresholds for data store updates, and to parameters that contribute to the determination of the name scores (SN_VAL, GN_VAL) and to the Composite Score of two record comparands.

4.13.4. Function

The HP functions as an independent data store with thresholds needed by the LFP and all the parameters needed by the HFS during the filtering process.

4.14. REFUSAL CODE CATEGORY DATA STORE DECOMPOSITION

4.14.1. Identification

This data store is known as the Refusal Code Level Data Store (RCL).

4.14.2. Type

4.14.2.1. It is recommended that the RCL be a parameter file, which can be accessed by the client so RC categories can be added to or changed with ease.

4.14.2.2. The RC data store will provide a list of the Refusal Codes and its Refusal Category, which is an indication of the level of seriousness of each Refusal Code.

4.14.2.3. The RCL will be referred to by the Hispanic Decision Matrix (HDM) and by the LFP and Hispanic Filter and Sorter.

4.14.2.4. The RCL has the following format:

Figure 73: Format: Refusal Code Level Data Store (RCL)

DATA FIELD	DATA TYPE	FIELD SIZE	DATA VALUE
REFUSAL CODE	alphanumerics	3	Standard Refusal Codes
REF_CAT	alphanumerics	3	RL0, RL1, RL2, RL3, RL4

Figure 74: Example: RCL (REF_CATs for example only)

REFUSAL CODE	REF_CAT
00	RL0
23	RL1
6C	RL2
07	RL3
G	RL4

4.14.2.5. Definitions

4.14.2.5.1. REFUSAL CODE: indicates each Visa Refusal Code (Codes and their Refusal Level (see VALUE) are for example only; they do not represent the complete list nor the accurate assignment of a Refusal Code to a Refusal Level).

4.14.2.5.2. REF_CAT: The RL# will appear in the form RL1, RL2, etc.

4.14.2.5.2.1. RL# is the Refusal Category to which a particular Refusal Code has been assigned. The Visa Office will assign Refusal Codes to one of 4 categories: RL1, RL2, RL3, RL4; RL0 is reserved for the Refusal Code 00. (The current distinction among Refusal Codes is a binary one: serious and non-serious. Assignment of Refusal Codes to more groups has not yet been done; the consequence is that one or more of these categories may not currently have a distinct value.) The RL# occurs in ascending order, from most serious to least serious Refusal Code. The RL# will be linked to a Year-of-

Birth Code (see Section 4.16) to determine the relevant subsets of records to be searched.

4.14.2.5.2.2. **RC0** refers to the Refusal Code 00

4.14.2.5.2.3. **RC1** refers to all Refusal Codes that have been designated as Type 1 Serious RC 1, i.e., the most serious, excluding 00.

4.14.2.5.2.4. **RC2** refers to all Refusal Codes that have been designated as Type 2 Serious RC, i.e., serious but less serious than RC0 and RC1.

4.14.2.5.2.5. **RC3** refers to all Refusal Codes that have been designated as Type 1 Non-Serious RC. These codes are less serious than the RC0, RC1 and RC2 codes.

4.14.2.5.2.6. **RC4** refers to Refusal Codes that have been designated as Type 2 Non-Serious. These codes are the least serious codes, less serious than the RC0, RC1, RC2 and RC3 codes.

4.14.3. Purpose

It has long been desirable to make more granular distinctions among the Refusal Codes. For many years, DOS has maintained a distinction between serious and non-serious codes; these different categories were correlated with different YOB search ranges. However, a mechanism for making greater distinctions will provide greater flexibility in delimiting the set to be retrieved during the first stage of record analysis, especially for Hispanic high frequency names, where more restricted retrievals are highly desirable. The introduction of five refusal code levels also provides the opportunity to correlate more year-of-birth ranges to the refusal code levels.

4.14.4. Function

The RCL provides information needed for the evaluation of record proximity in the Hispanic filtering process and contributes to the delimitation of database records retrieved through the RLYOB Data Store.

4.15. YEAR-OF-BIRTH RANGE DATA STORE DECOMPOSITION

4.15.1. Identification

This data store is known as the Year-of-Birth Range Data Store (YR).

4.15.2. Type

4.15.2.1. It is recommended that the YR be a parameter file, which can be accessed by the client so YOB ranges can be set. Alternatively, it could be represented as a system parameter whose value(s) are set in an .ini file.

4.15.2.2. The YR will define the YOB ranges that will be associated with a Refusal Level (see Section 4.16).

4.15.2.3. This data store has the following format:

Figure 75: Format: Year-of-Birth Range Data Store (YR)

DATA FIELD	DATA TYPE	FIELD SIZE	VALUE	DATA DEFINITION
YOB0	integer	1	0	exact date of birth
YOB1	character	1	A	exact year, inverted month and day
YOB2	character	1	B	exact year of birth
YOB3	integer	2	1...99	narrow year of birth range
YOB4	integer	2	1...99	standard year of birth range
YOB5	integer	2	1...99	wide year of birth range
YOB6	integer	2	1...99	unlimited year of birth range

4.15.2.4. Definitions

4.15.2.4.1. YOB# is the Year-of-Birth Range category whose value indicates the year-of-birth range to be searched. The year-of-birth VALUE indicates the search range, that is, the number of years on either side of a given year-of-birth to be searched. For example, if the input year is 1962 and YOB3 range is 4, the search will cover a range of nine years, 1958-1966. The range includes the full year, so all of 1958 and all of 1966.

4.15.2.4.1.1. There are seven YOB# categories, YOB0, YOB1, YOB2, YOB3, YOB 4, YOB5, YOB6.

- **YOB0** is a single integer that refers to an exact month, day, year of birth. If YOB0 is specified, the system must be able to match the month, day and year of the Date of Birth of an input record and a database record.
- **YOB1** is a single character (A) that refers to an exact year-of-birth with the month and day inverted.
 - If YOB1 is specified, the system must be able to match the year of Date of Birth and an inverted month and day (DEC 03 → MAR 12) of the input record and the database record.
 - YOB1 will be relevant to the Hispanic Filter and Sorter, but may not function as a search

parameter since the value would be subsumed in YOB2.

- **YOB2** is a single character (B) that refers to an exact year-of-birth. If YOB2 is specified, the system must be able to match the year of the Date-of-Birth of an input record and a database record.
- **YOB3** is a one- or two-place integer (1...99) that refers to a narrow year-of-birth range. Narrow year-of-birth range is usually defined as 1 year (for a search range of 3 years).
- **YOB4** is one- or two-place integer (1...99) that refers to a standard year-of-birth range. Standard year-of-birth range is usually defined as 3 years (for a search range of 7 years).
- **YOB5** is a one- or two-place integer (1...99) that refers to a wide year-of-birth range. Wide year-of-birth range is usually defined as 5 years (for a search range of 11 years).
- **YOB6** is a one- or two-place integer (1...99) that refers to an unlimited or extremely wide year-of-birth range. Unlimited year-of-birth range would be set sufficiently high to include all (or all desired) years-of-birth in the database (e.g., 50).

4.15.3. Purpose

This YR provides a greater granularity in the year-of-birth range and, therefore, greater flexibility in delimiting the set to be retrieved during the first stage of record analysis. The correlation of five refusal code levels to different year-of-birth ranges will help to delimit the number of records to be searched and to define the more valuable set of records. For the Hispanic processor, greater precision in the year-of-birth range is especially important in the High Frequency Processor where more restricted retrievals are highly desirable.

4.15.4. Function

- 4.15.4.1. The YR permits greater granularity in the Date-of-Birth types related to the system.
- 4.15.4.2. The YR will be accessed by the Refusal Code Level/YOB Range Data Store, which will limit the retrieval range in the Hispanic Search Engine.
- 4.15.4.3. The YR data store will define the YOB ranges referred to in the Hispanic Decision Matrix (HDM).
- 4.15.4.4. The YR will contribute to the Hispanic Filter and Sorter to contribute information to the composite score.

4.16. REFUSAL CODE LEVEL / YOB RANGE DATA STORE MODULE DECOMPOSITION

4.16.1. Identification

This data store is known as the Refusal Code Level/YOB Range Data Store (RLYOB).

4.16.2. Type

4.16.2.1. The RLYOB is a matrix that merges the values in the Refusal Code Level (RCL) Data Store and the Year-of-Birth Range (YR) Data Store.

4.16.2.2. For each Refusal Level (RL), a Year-of-Birth (YOB) Range is specified.

4.16.2.2.1. Only one YOB Range for each RL is permitted.

4.16.2.2.2. The same YOB Range may apply to more than one RL.

4.16.2.3. The RLYOB has the following format:

Figure 76: Format: Refusal Level/Year-of-Birth Range Data Store (RLYOB)

DATA FIELD	DATA TYPE	FIELD SIZE	VALUE RANGE	DATA VALUE
RL#	character	3	RL0...4	RL0, RL1, RL2, RL3, RL4
YOB#	character	4	YOB0...6	YOB0, YOB1, YOB2, YOB3, YOB4, YOB5, YOB6

Figure 77: Example: RLYOB Data Store

RL#	YOB#
RL0	YOB5
RL1	YOB4
RL2	YOB3
RL3	YOB3
RL4	YOB2

4.16.2.4. Definitions:

4.16.2.5. RL#: is a character string that indicates the Refusal Level of the Refusal Code.

4.16.2.6. YOB#: is a character string that indicates the Date-of-Birth Range Category of the comparands.

4.16.3. Purpose

Retrieval of records from the database should be delimited by a relationship between the Refusal Code Level and the Year-of-Birth Range. It will restrict the number of records to be reviewed.

4.16.4. Function

The RLYOB is a resource for the Hispanic Search Engine to delimit the LF records retrieved from the database.

4.17. COUNTRY-OF-BIRTH PROXIMITY DATA STORE

4.17.1. Identification

This module is known as the Country-of-Birth Proximity Data Store (COBPROX).

4.17.2. Type

4.17.2.1. The COBPROX is a data store whose cells contain a decimal value that reflects the degree of relationship between the country represented on two country-of-birth.

4.17.2.2. The COBPROX has the following format:

Figure 78: Design: COBPROX Data Store

DATA FIELD	DATA TYPE	FIELD SIZE	VALUE RANGE	DATA VALUE
COB#1	character	4	alphabetics	COB Code
COB#2	character	4	alphabetics	COB Code
COBVAL	decimal	4	0.00...1.00	Various

Figure 79: Example: Piece of COBPROX Data Store (COBVAL for example only)

COB#1	COB#2	COBVAL
AGS	AGS	1.00
AGS	GRBR	0.05
AGS	VTNM	0.05
AGS	MORO	0.05
AGS	SYR	0.05
ALG	ALG	1.00
ALG	MORO	0.85
ALG	GRBR	0.05
ALG	VTNM	0.05
MORO	MORO	1.00
MORO	GRBR	0.05
MORO	VTNM	0.05
GRBR	GRBR	1.00
GRBR	VTNM	0.05
VTNM	VTNM	1.00

4.17.2.3. Definitions:

4.17.2.3.1. COB#1: is the 4-character Country-of-Birth Code of one of the comparands.

4.17.2.3.2. COB#2: is the 4-character Country-of-Birth Code of one of the comparands.

4.17.2.3.3. COBVAL: is the decimal value assigned through the HCOB and other COB Category Data Stores that are culture-specific (as they are developed). A default value will be assigned for those COBs that do not enter into special relations. The COBVAL indicates the degree of relationship between the two COBs.

4.17.3. Purpose

The COBPROX Data Store provides information on the relative value of the COBs in two comparands. This value can serve to limit the COBs that are accessed for retrieval.

4.17.4. Function

The COBPROX is populated by the HCOB and any other partition-specific Country-of-Birth Category Data Stores. The COBPROX provides COB relationship information.

4.18. HISPANIC COUNTRY-OF-BIRTH CATEGORY DATA STORE DECOMPOSITION

4.18.1. Identification

This data store is known as the Hispanic Country-of-Birth Category Data Store (HCOB).

4.18.2. Type

This HCOB is a data store that will serve as the source of information for the COBPROX Data Store, populating the COBVAL, and will provide the COB Category (COBCAT) necessary for the Hispanic Filter and Sorter.

Figure 80: Design: Hispanic Country-of-Birth Category Data Store (HCOB)

DATA FIELD	DATA TYPE	FIELD SIZE	VALUE RANGE	DATA VALUE
COB#1	characters	4	alphabetic	COB Code
COB#2	characters	4	alphabetic	COB Code
COBCAT	characters	5	alphanumeric	COB1...COB99
COBVAL	decimal	4	0.00...1.00	Various

Figure 81: Example: Piece of HCOB (Values for example only.)

COB#1	COB#2	COBCAT	COBVAL
AGS	AGS	COB1	1.00
AGS	MEX	COB2	0.95
AGS	COL	COB4	0.65
AGS	ENE	COB4	0.65

AGS	PORT	COB5	0.60
COL	COL	COB1	1.00
COL	MEX	COB4	0.65
COL	VENE	COB3	0.85
COL	PORT	COB5	0.60
MEX	MEX	COB1	1.00
MEX	VENE	COB4	0.65
MEX	PORT	COB5	0.60
VENE	VENE	COB1	1.00
VENE	PORT	COB5	0.60
PORT	PORT	COB1	1.00
...			

4.18.3. Definitions

4.18.3.1. COB#1: is the 4-character COB Code of one of the comparands.

4.18.3.2. COB#2: is the 4-character COB Code of one of the comparands.

4.18.3.3. COBCAT: is the category assigned to the relationship of two COBs.

4.18.3.3.1. Categories might be defined as Exact, State, Geographic Region, Dialect Region.

4.18.3.3.2. All relationships are adjustable.

4.18.3.3.3. Example COB Categories are:

- **COB1: Exact** represents an exact match of the COBs: AGS/AGS; the COBPROXVAL would be 1.00.
- **COB2: State Relationship** represents the set of COBs that are states within one country (currently only the Mexican States qualify). The score would be something less than that applied to an exact match but nonetheless high: 0.95.
- **COB3: Northern South America** represents the set of COBs that are in close geographic proximity and share naming conventions: COL/VENE. The value assigned would be less than that for COB2: 0.85.
- **COB4: All Latin America** refers to all COBs in Central and South America and the Spanish-speaking Caribbean. The value assigned would be less than that for COB2: 0.65.
- **COB5: Similar** refers to COBs that have qualified as Hispanic but may not exhibit Hispanic naming conventions: Brazil, Portugal.
- **COB6: All** refers to all COBs and is assigned a value that will allow the search of all COBs; it would be the lowest decimal value used.

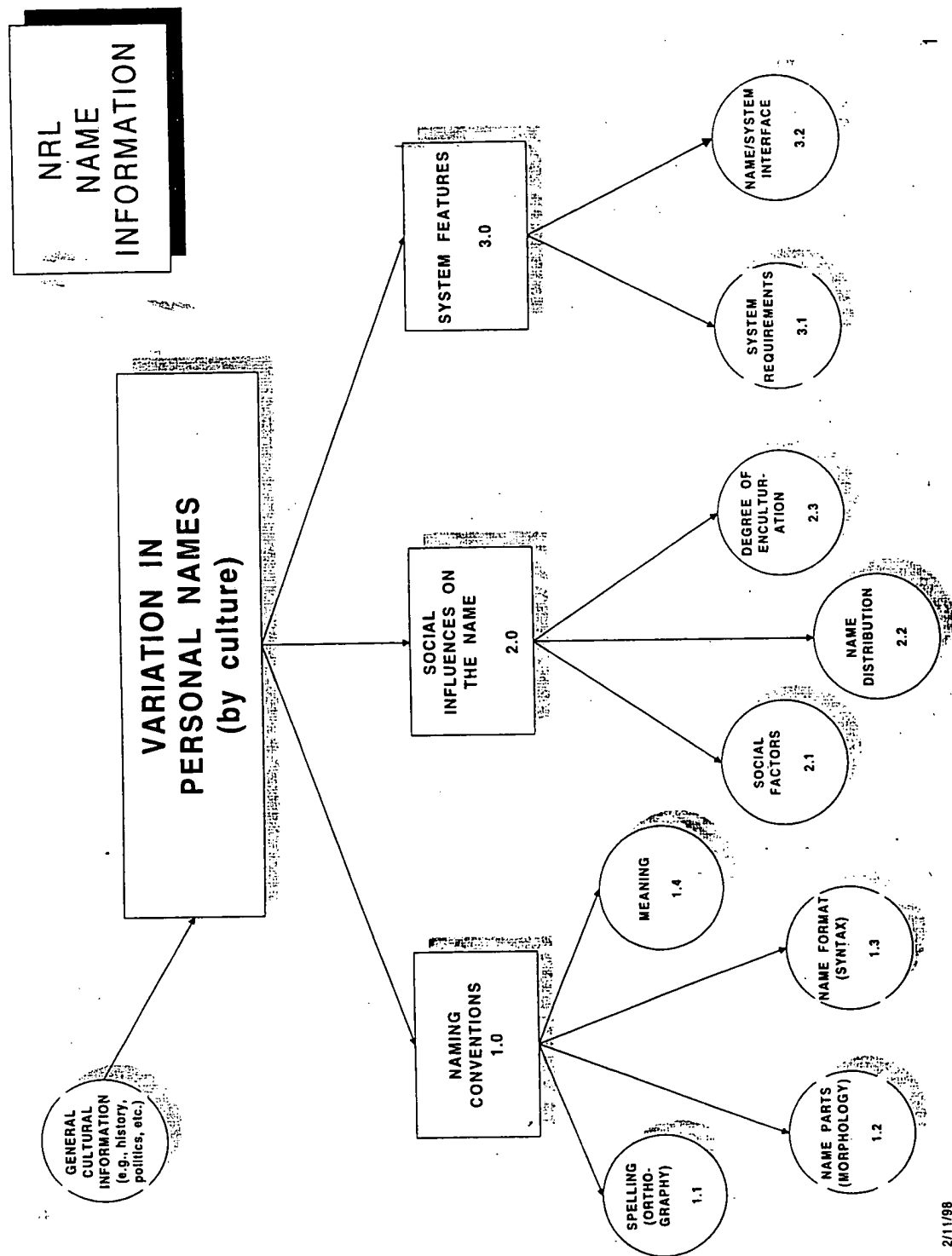
4.18.3.4. COBVAL: is the decimal value that will be assigned to a particular COB relationship; this value will be used to determine the COBs that will be permitted in the retrieval process.

4.18.4. Purpose

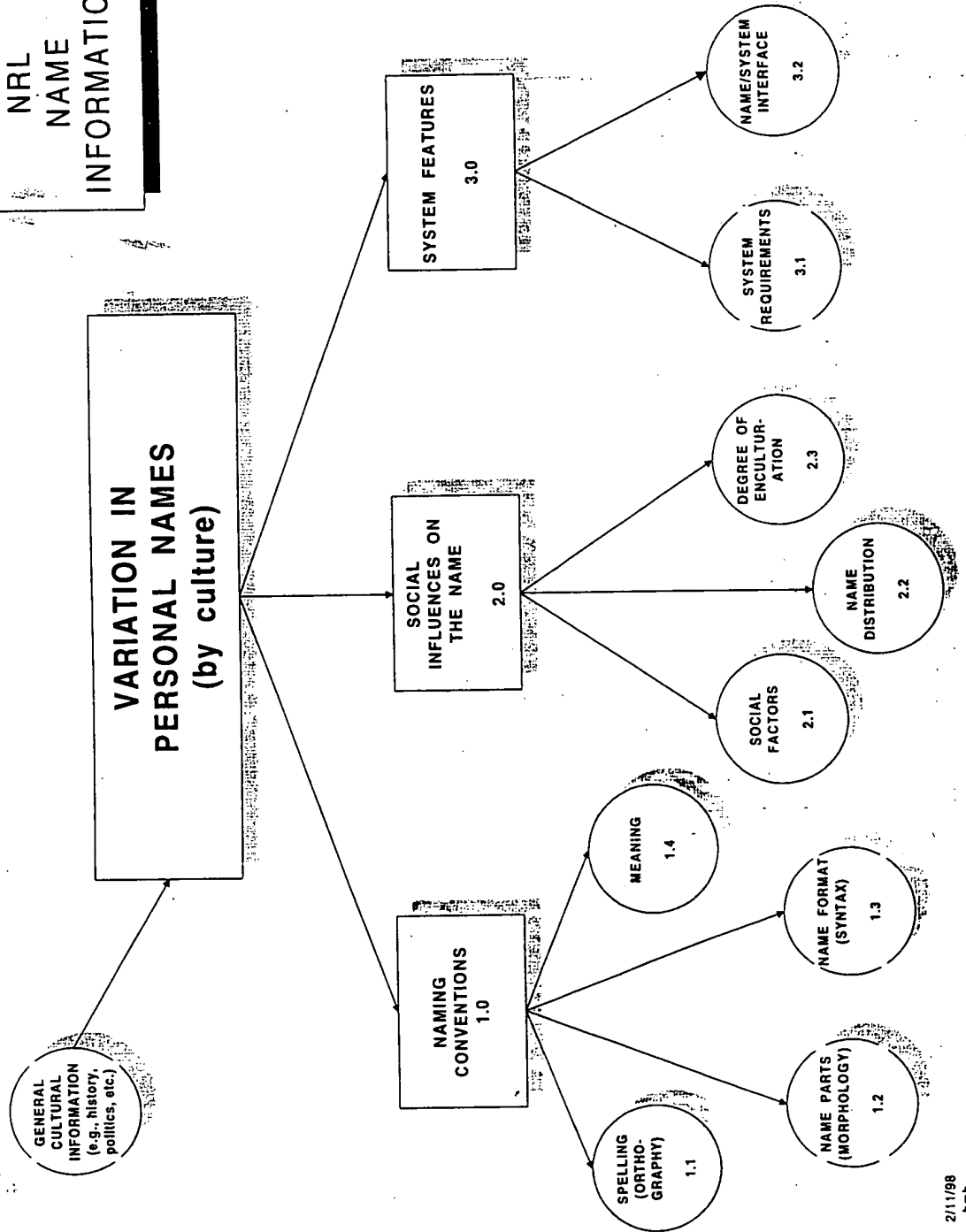
Pre-defined COB category relationships will provide a definition of the values that appear in the COBPROX Data Store.

4.18.5. Function

These COB categories will provide information about COB relationships that will contribute to determination of the Composite Score in the Arabic Filter and Sorter.

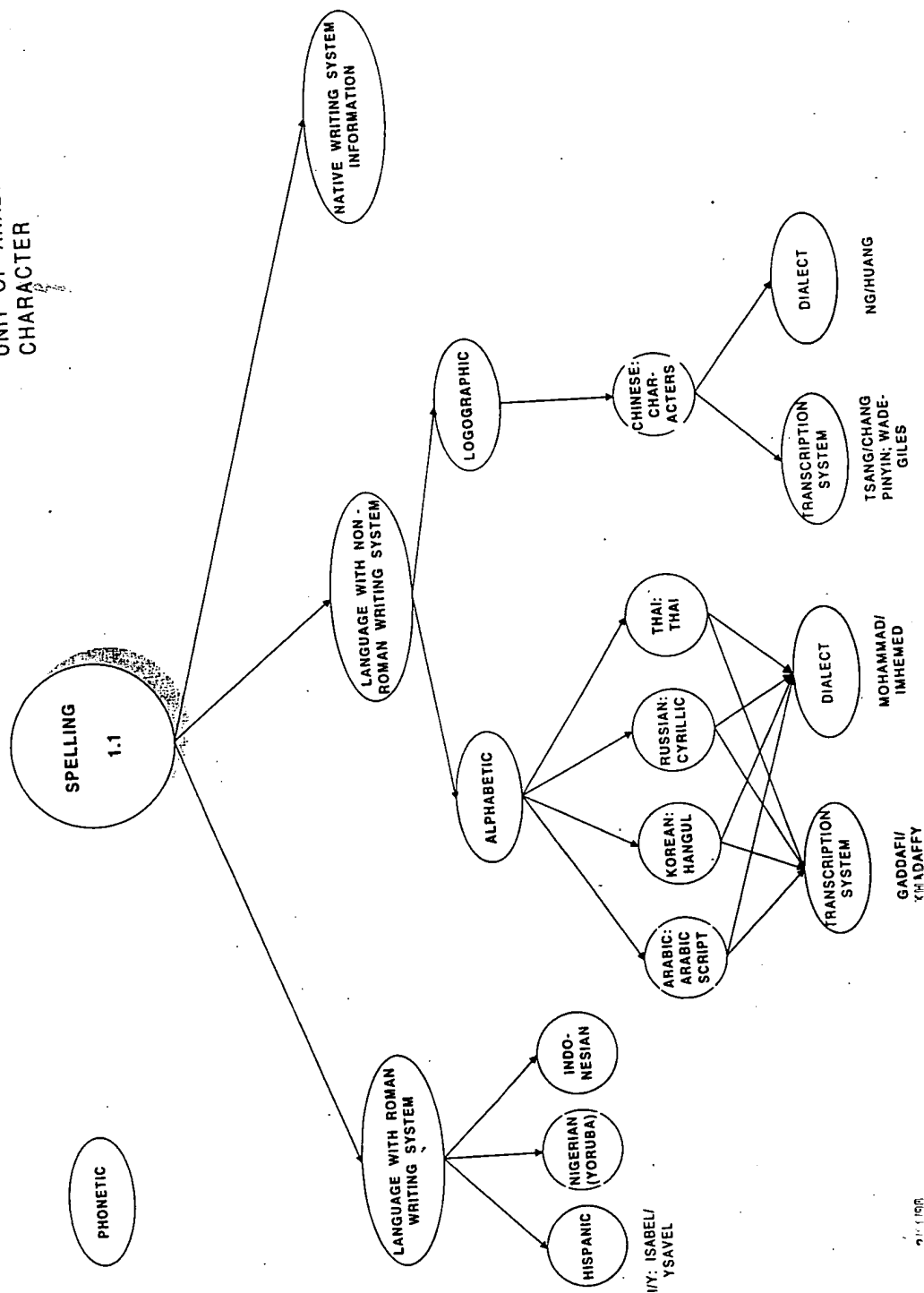


NRL
NAME
INFORMATION



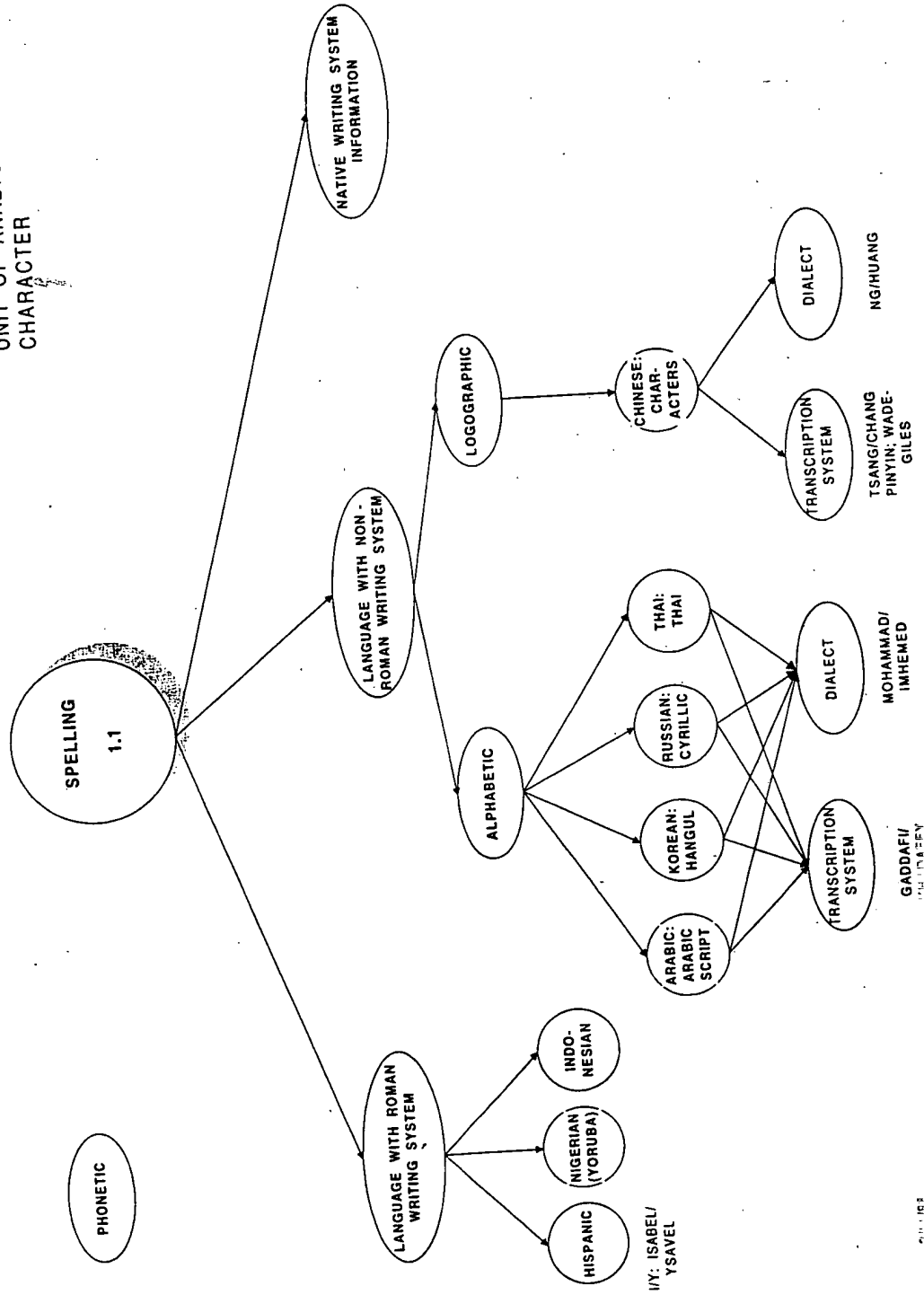
ORTHOGRAPHY

UNIT OF ANALYSIS:
CHARACTER



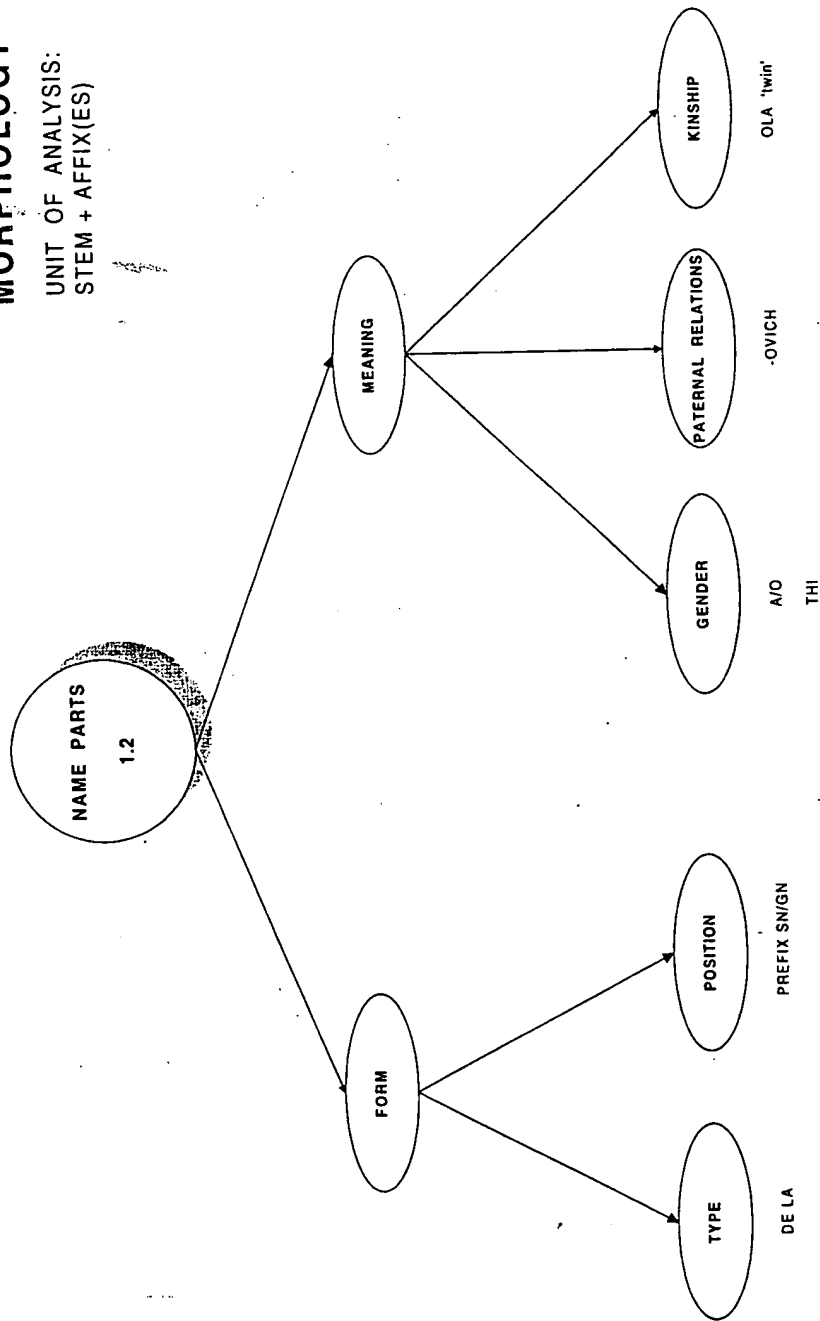
ORTHOGRAPHY

UNIT OF ANALYSIS:
CHARACTER



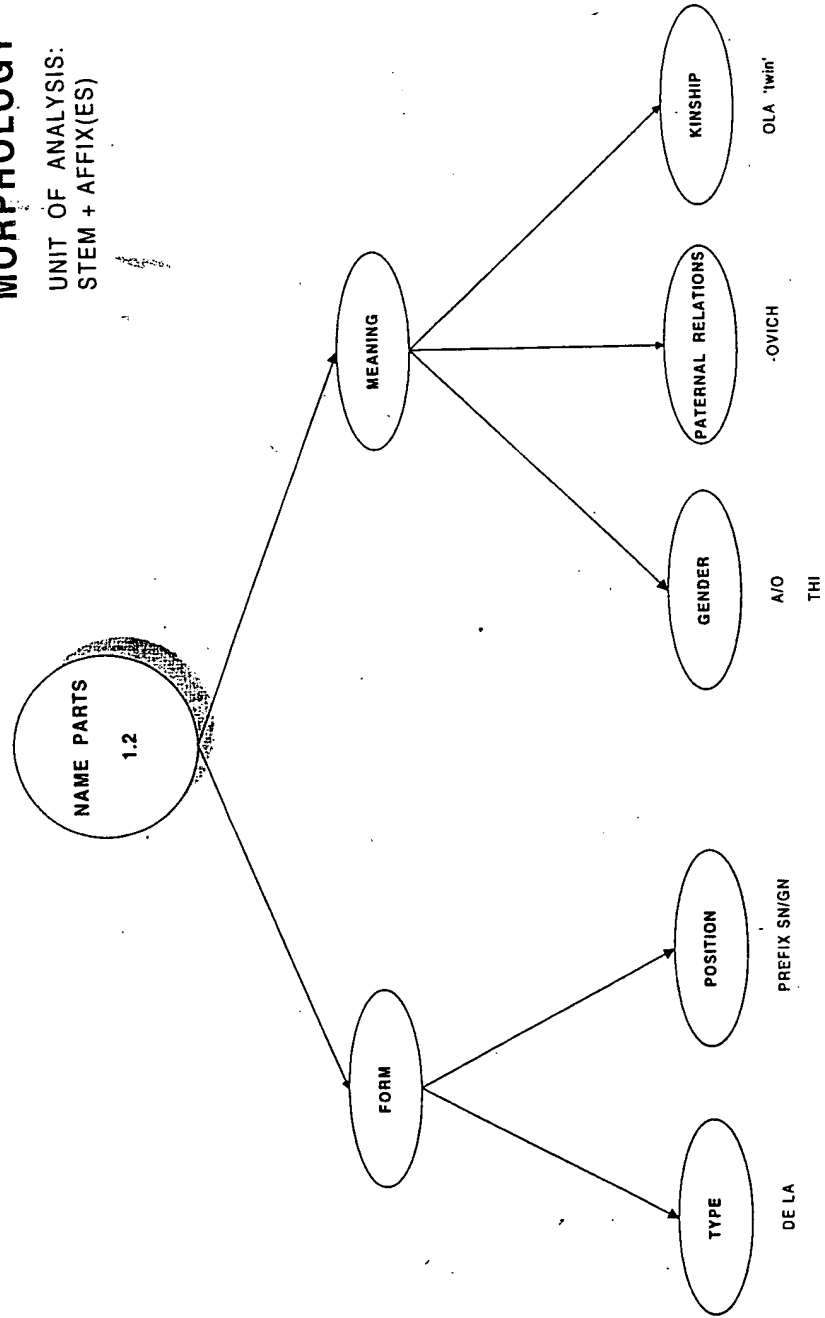
MORPHOLOGY

UNIT OF ANALYSIS:
STEM + AFFIX(ES)



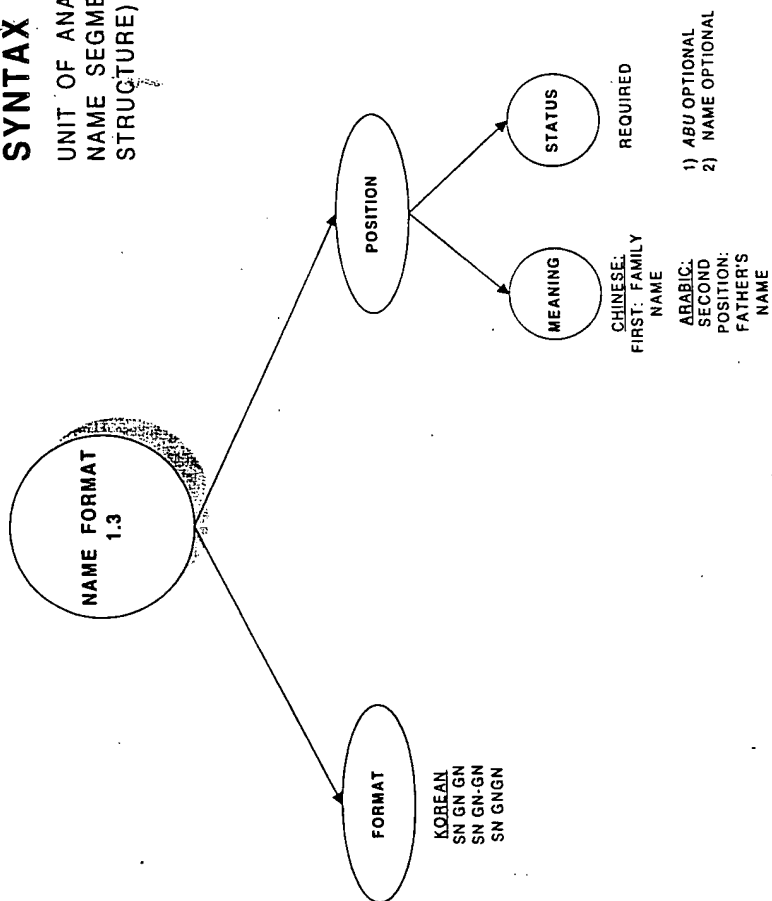
MORPHOLOGY

UNIT OF ANALYSIS:
STEM + AFFIX(ES)



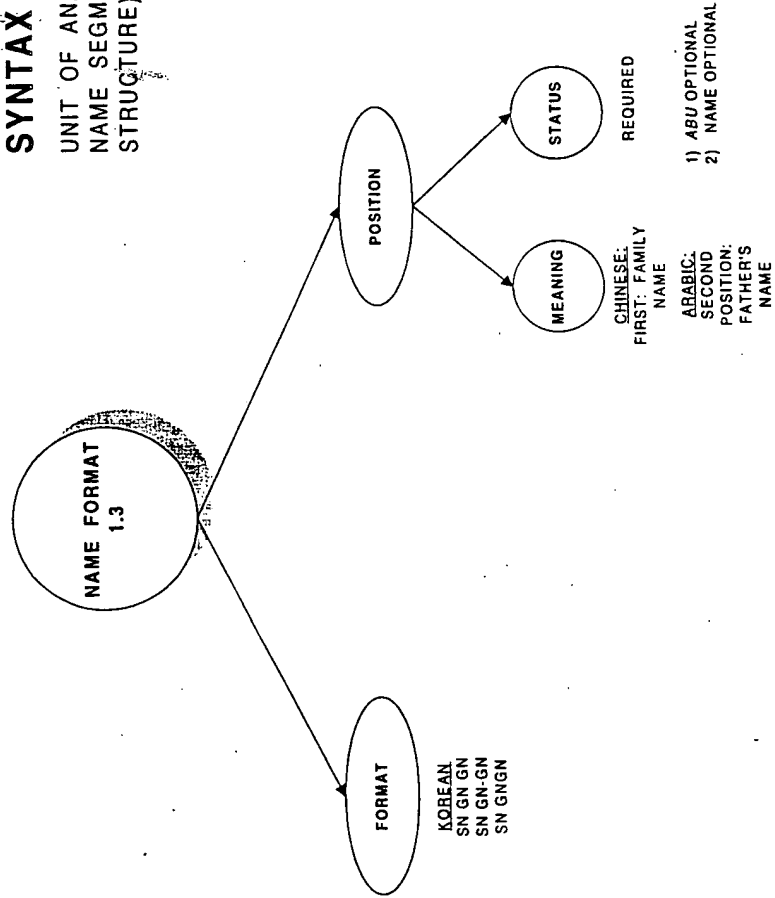
SYNTAX

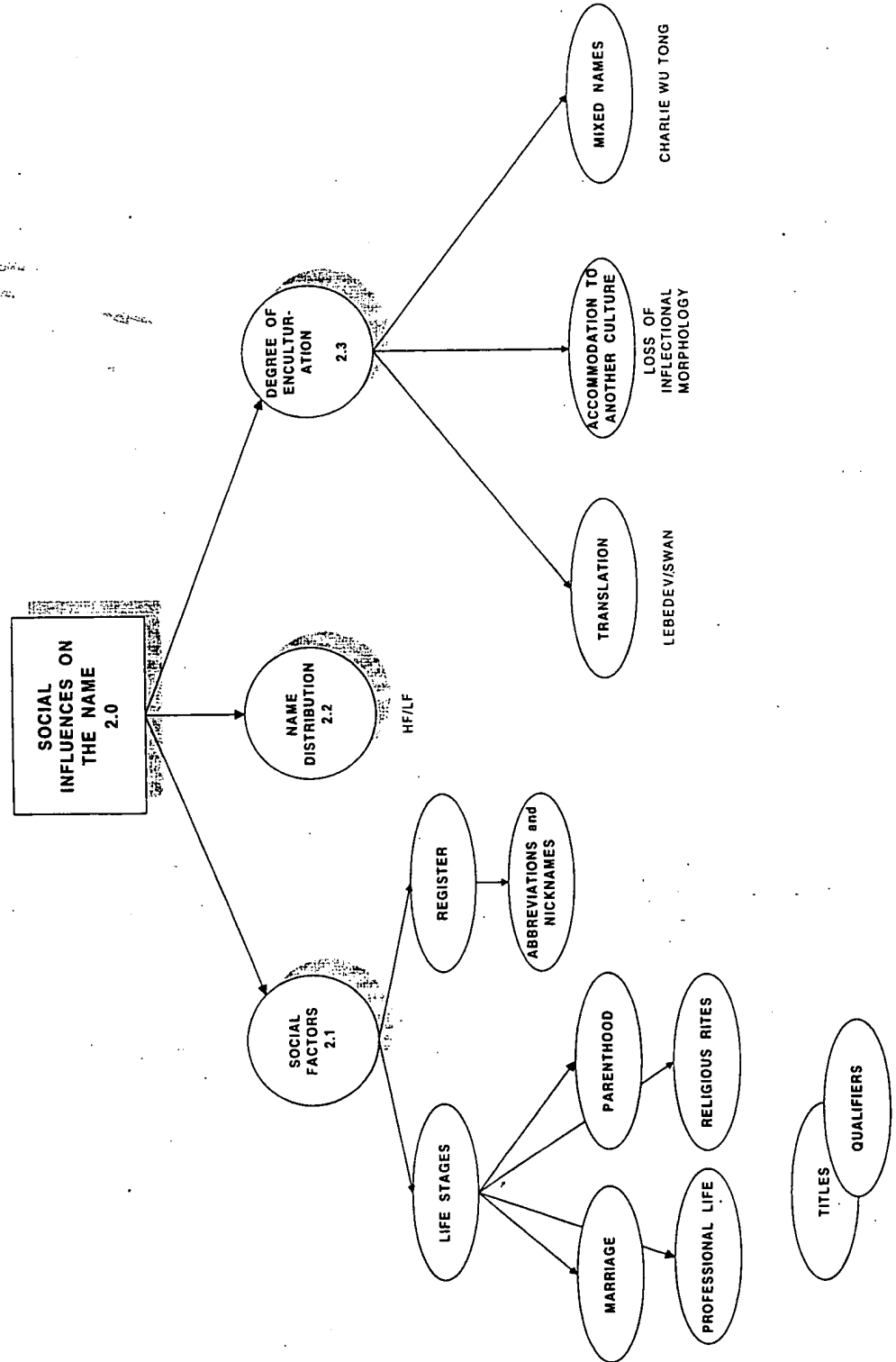
UNIT OF ANALYSIS:
NAME SEGMENT (NO INTERNAL
STRUCTURE)

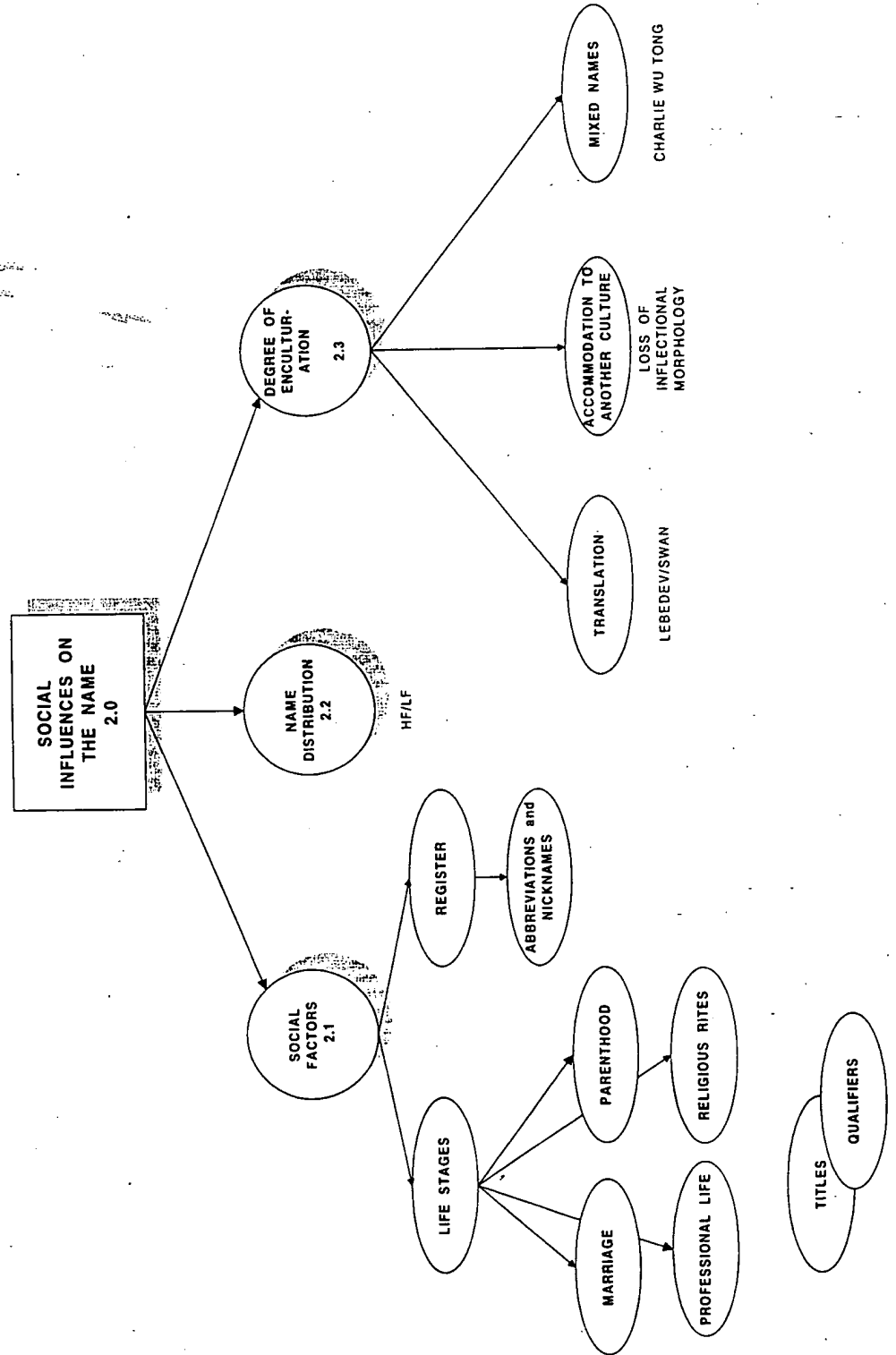


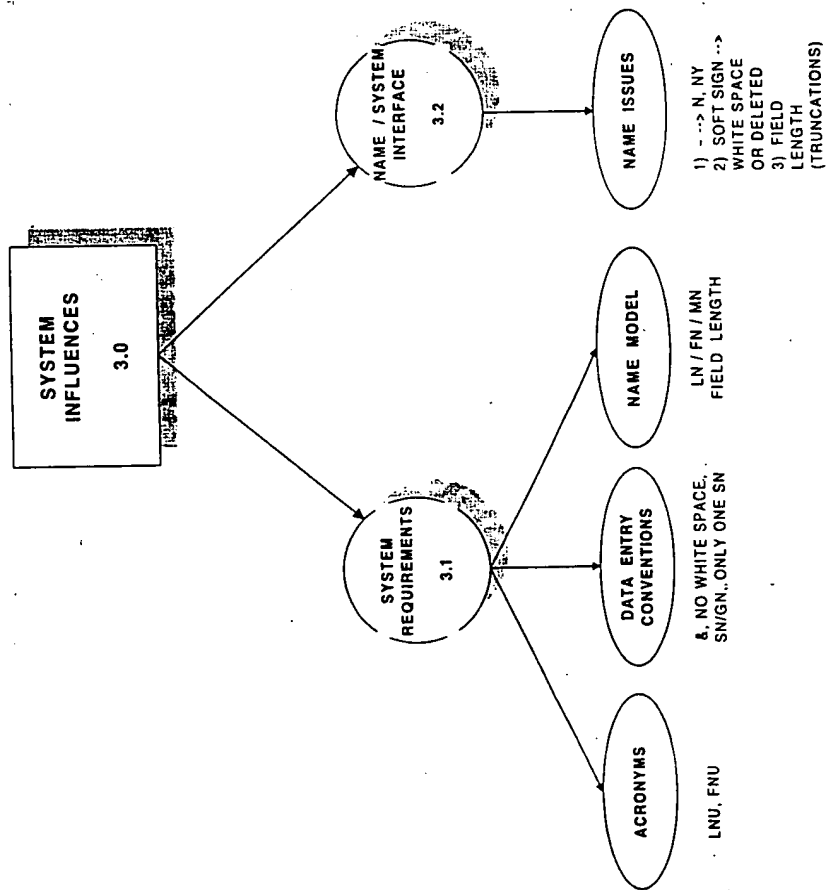
SYNTAX

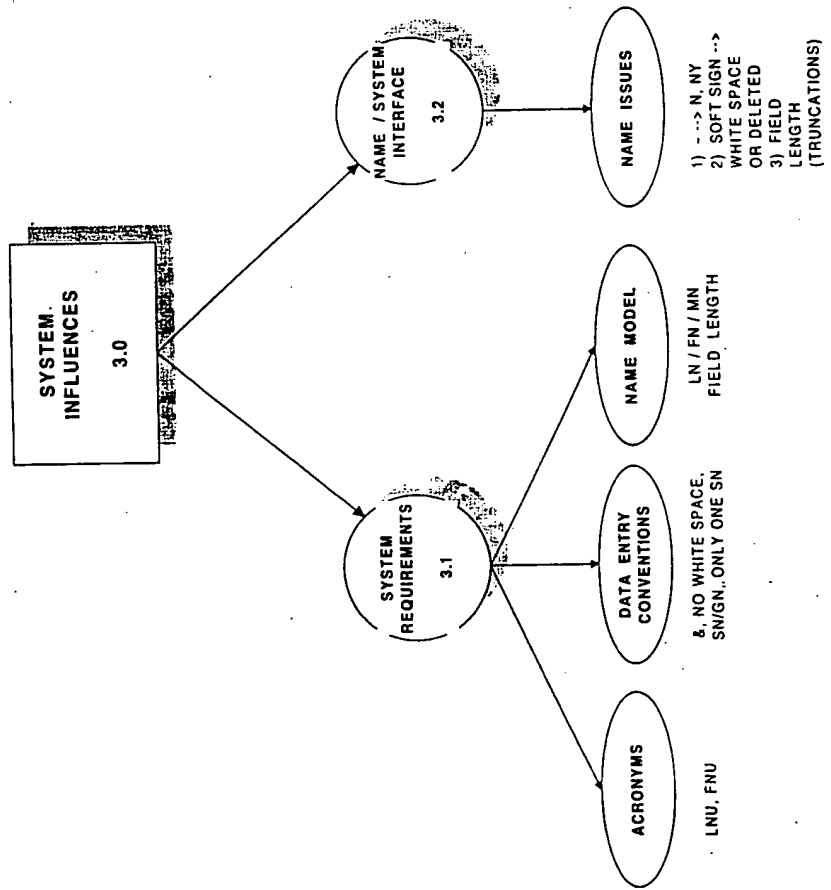
UNIT OF ANALYSIS:
NAME SEGMENT (NO INTERNAL
STRUCTURE)











U.S. Department of State
Bureau of Consular Affairs

Consular Lookout and Support System—Enhanced (CLASS-E)



Presentation to
CA / EX / CSD

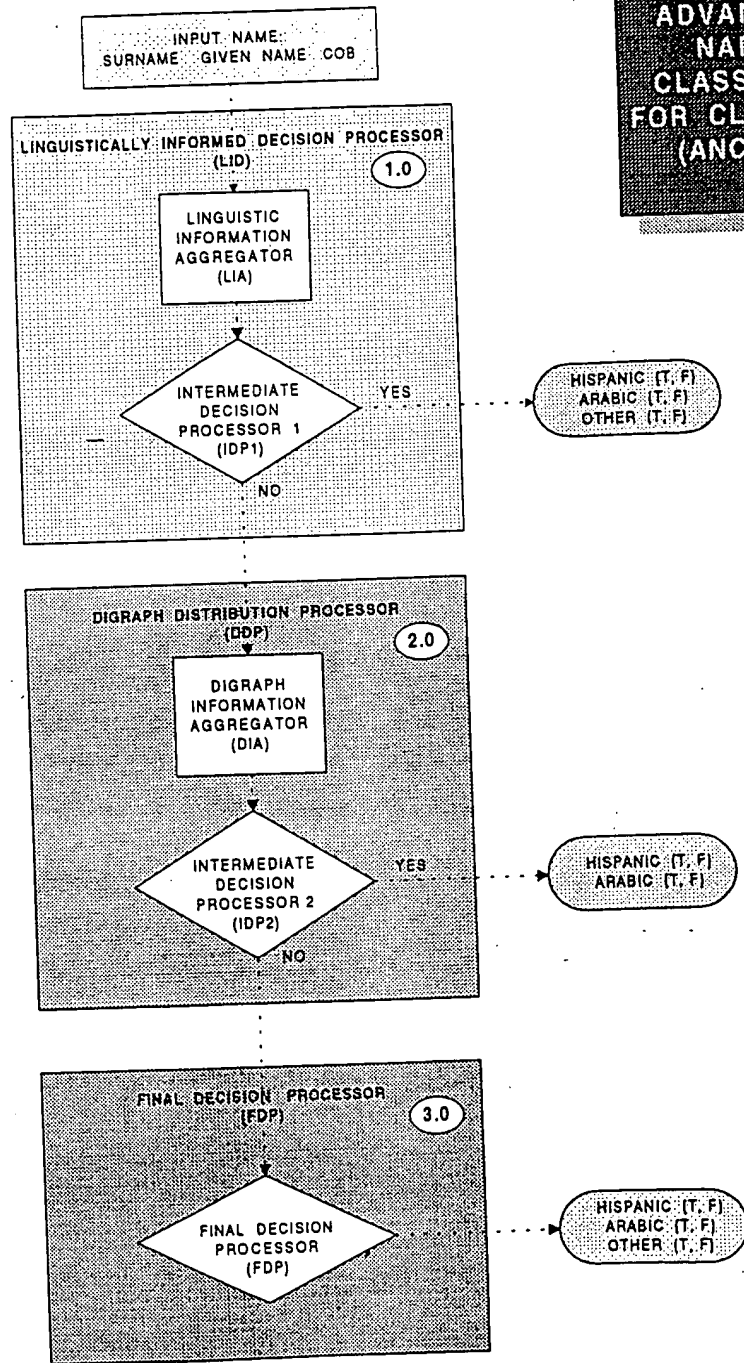
Advanced Name Classifier for CLASS-E (ANC-E)
and
Hispanic Name Search Algorithm for CLASS-E (HNA-E)

September 18, 1997

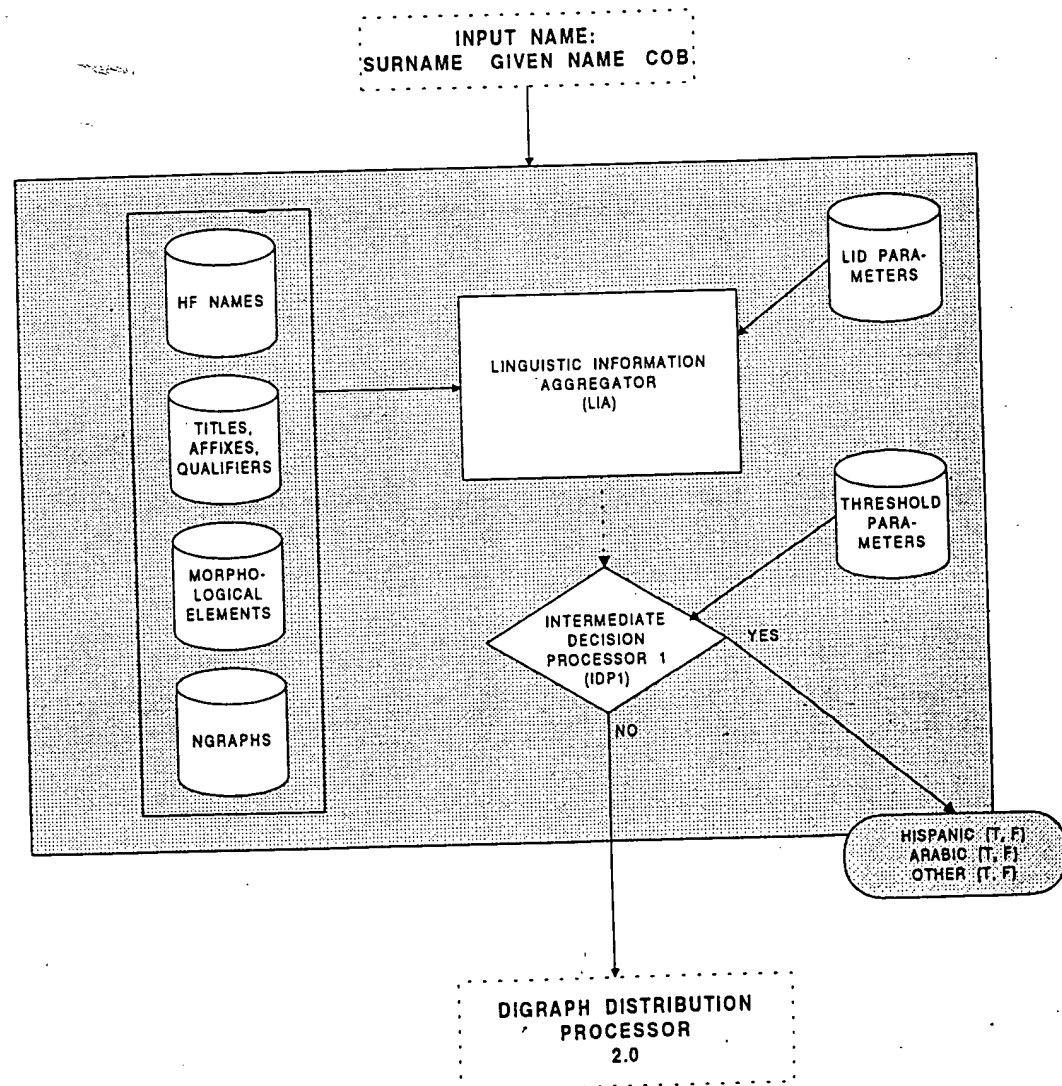


Language Analysis Systems, Inc.
2214 Rock Hill Road—Herndon, VA—20170

**ADVANCED
NAME
CLASSIFIER
FOR CLASS - E
(ANC - E)**

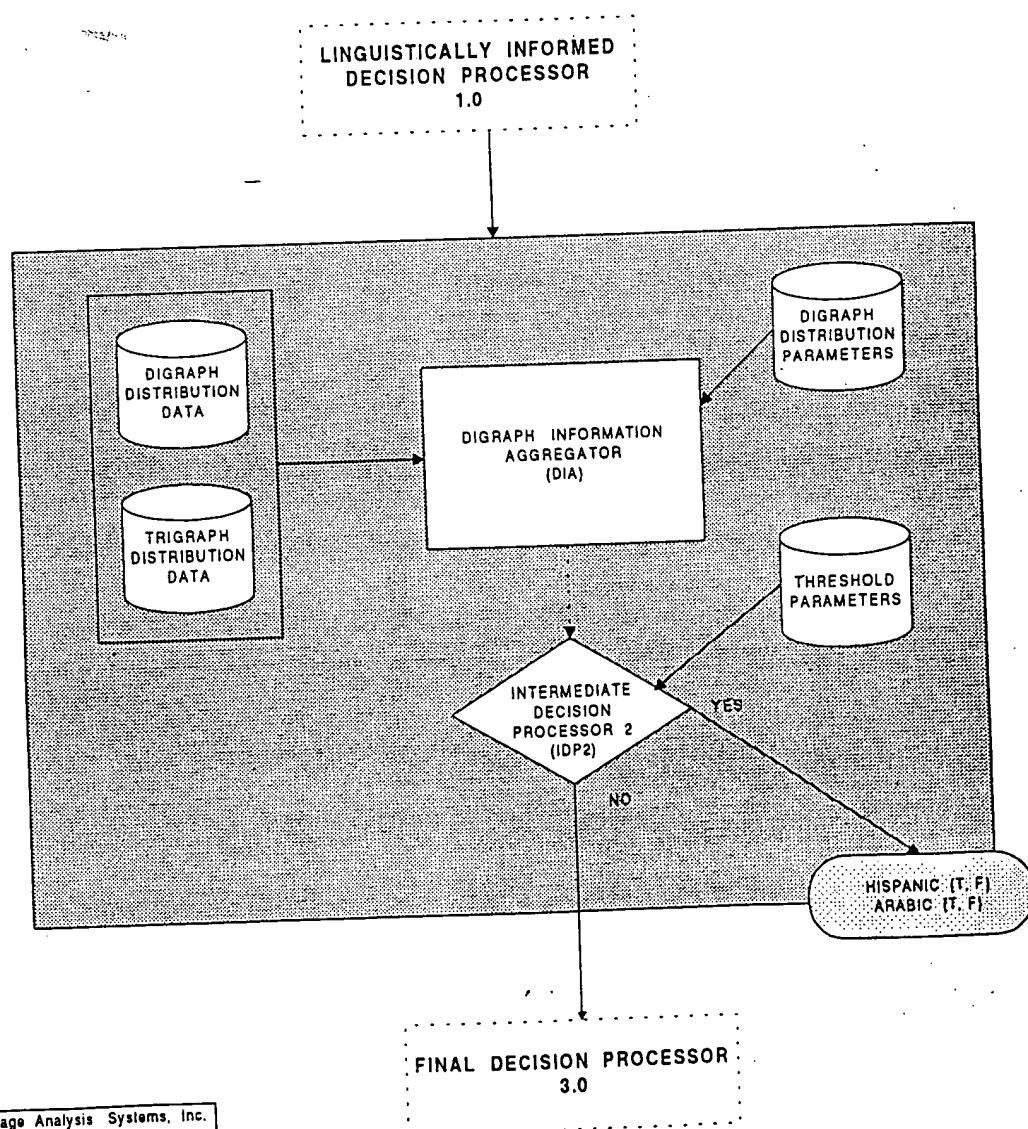


LINGUISTICALLY INFORMED DECISION PROCESSOR (LID)



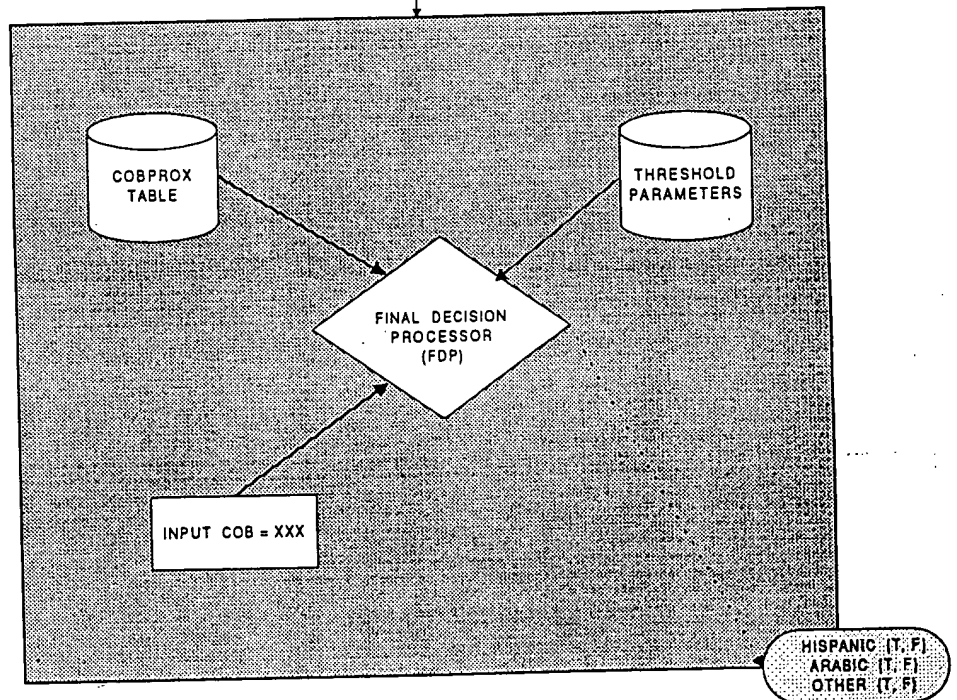
2.0

DIGRAPH DISTRIBUTION PROCESSOR (DDP)



FINAL DECISION PROCESSOR (FDP)

DIGRAPH DISTRIBUTION
PROCESSOR
2.0



LINGUISTICALLY INFORMED DECISION PROCESSOR (LID)

Factors:		In Field: 10 OutField: 8	In Field: 8 OutField: 6	In FieldSN: 5 OutFieldSN: 3 In FieldGN: 4 OutFieldGN: 2	In FieldSN: 3 OutFieldSN: 2 In FieldGN: 2 OutFieldGN: 1	N-Grams		Morphology	
High Frequency SN	High Frequency GN	Prefixes	N-Grams	Morphology					
H S Garcia 3	H G Jose 3	H S de 1	H S -ndez 3	A S adin 3					
H S Salazar 2	H G Francisco 2	H S la 1	H S -guez 2	A S edidin 2					
H S Sanbrano 1	H G Mario 1	H S las 1	H S -llo 1	A S uddin 1					
O S Greco 5	O B Luigi 3	O B il 1	O S -llo 3	O S ento 5					
O B Giuliano 2	O G Antonio 2	O B el 1	O B -ini 2	O S etti 4					
O S Silvestri 1	O G Adalberto 1	O B lo 1	O S -agio 1	O S ini 2					

N.B.: The data shown here are for the purpose of illustration only and do not necessarily reflect actual table values.

N.B. The data shown here are for the purpose of illustration only and do not necessarily reflect actual trading volumes.

DELGADILLO DE GARCIA, JOSE ANTONIO

	-ILLO (3*1)	ANTONIO -- (8*2)	DE (5*1)	GARCIA (10*3)	JOSE (8*3)	
Hispanic:	--		--	--	--	= 62
Arabic:	--		--	--	--	= 0
Other:	(3*3)		--	--	--	= 25

LID Threshold: Hispanic 65; Arabic 57; Other 56

Hispanic: F; Arabic: F; Other: F
(N.B. Values for illustration only.)

DIGRAPH DISTRIBUTION PROCESSOR (DDP)

DELGADILLO DE GARCIA, JOSE ANTONIO

DIGRAPHS	
A EZ	-1.0422
A NT	22.8733
A BD	38.7221
H BD	1.0372
H EZ	42.3947
H RI	16.1242

TRIGRAPHS	
A EZ#	-10.0422
A #NT	-48.1743
A BD#	48.4551
H BD#	-32.1742
H EZ#	47.5327
H #RI	11.1242

Digraph/Trigraph Scores:

HISPANIC: 44.2331
ARABIC: -32.8765

DI_Threshold: Hispanic 45.1116; Arabic 1.4532
Hispanic: F; Arabic: F

FINAL DECISION PROCESSOR (FDP)

DELGADILLO DE GARCIA, JOSE ANTONIO
COB = COL

COB	PART	COB2
COL	H	COL
COL	H	VENE
COL	H	BOL
EGYP	A	EGYP
EGYP	A	UAE

UNDER_DL_THRESHOLD: Hispanic 5; Arabic 10
UNDER_LID_THRESHOLD: Hispanic 6; Arabic 8; Other 3

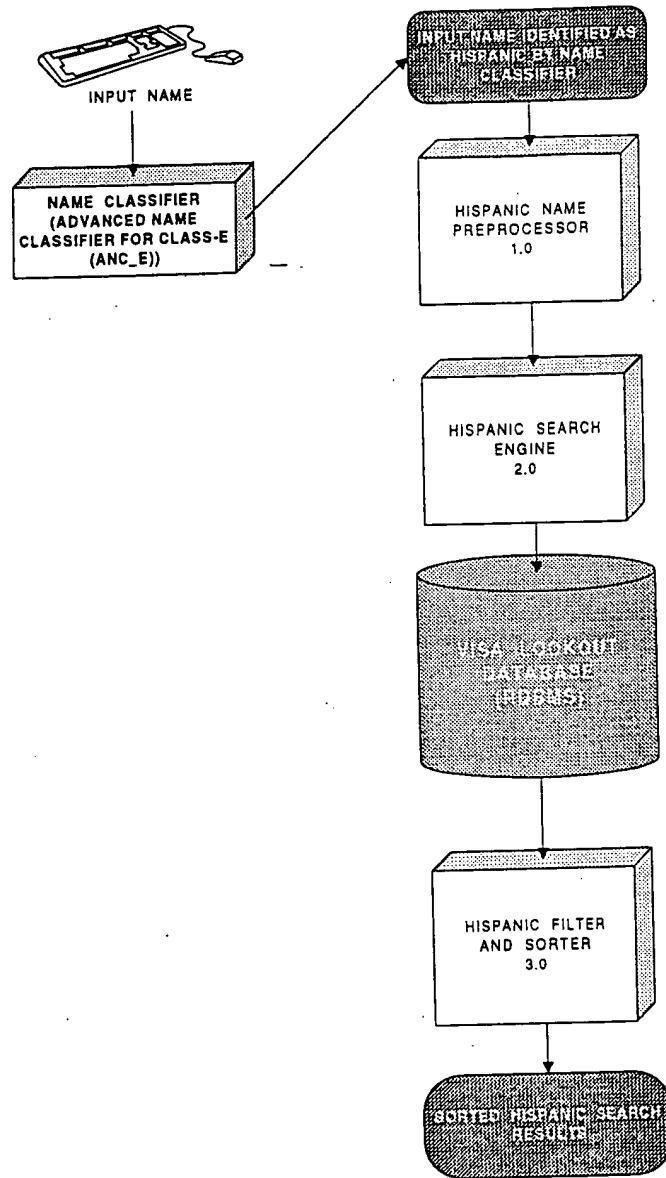
HISPANIC → YES
UNDER_DL_THRESHOLD -- yes
UNDER_LID_THRESHOLD -- yes
COB - H

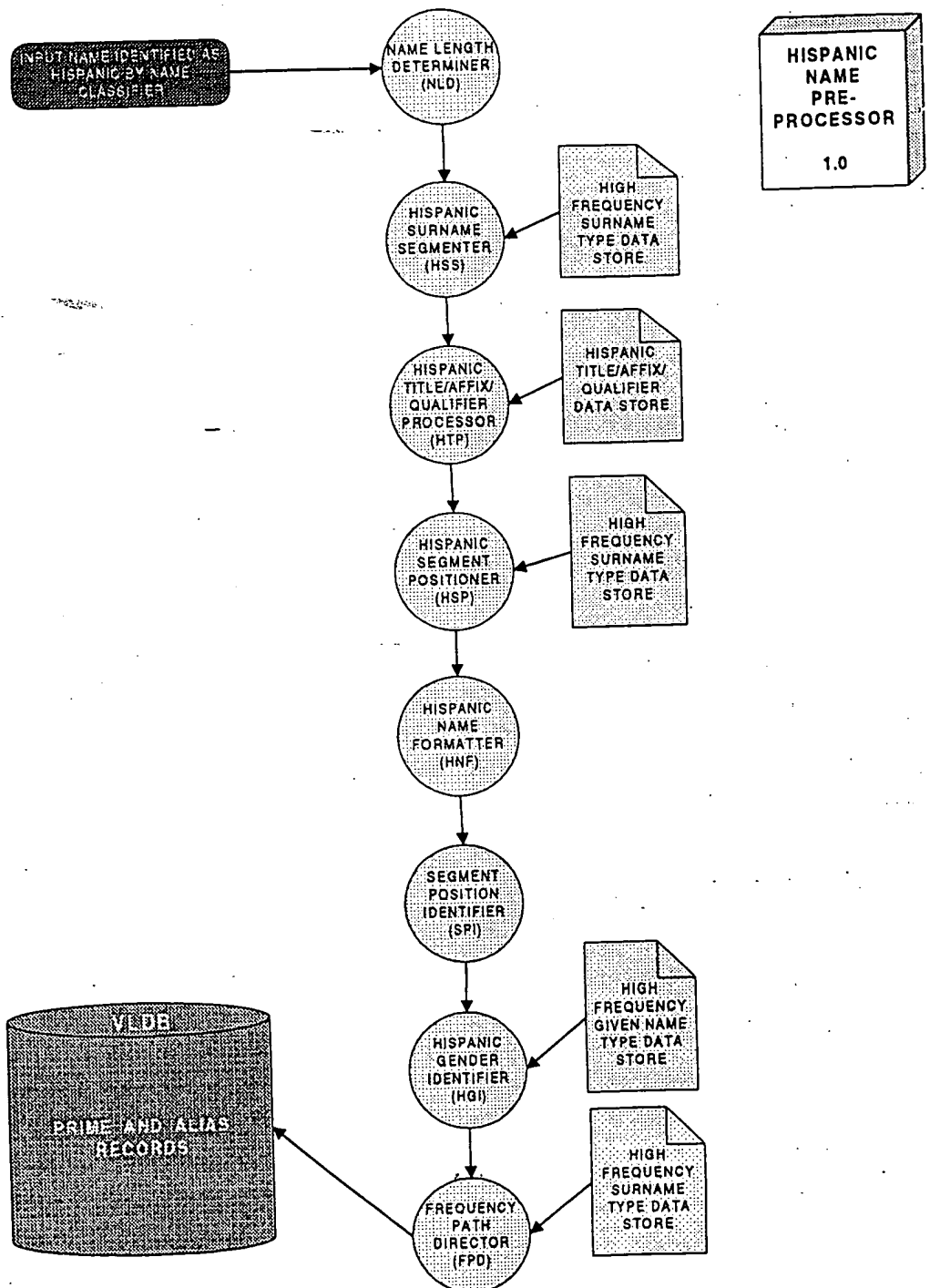
ARABIC → NO
UNDER_DL_THRESHOLD -- no
UNDER_LID_THRESHOLD -- no
COB - H

OTHER → NO
UNDER_LID_THRESHOLD -- no
COB - H

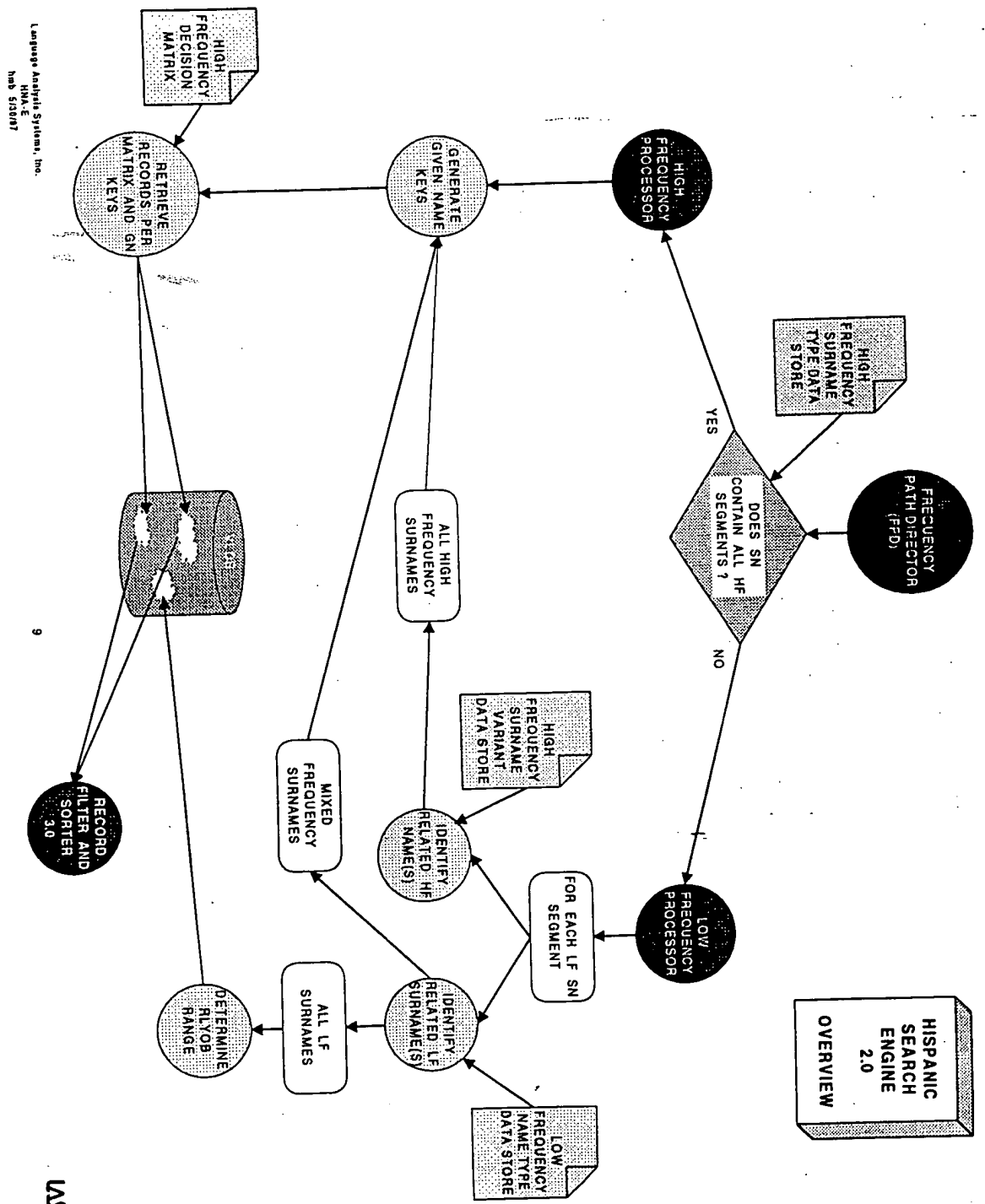
ADVANCED HISPANIC NAME SEARCH ALGORITHM for CLASS - E (HNA-E)

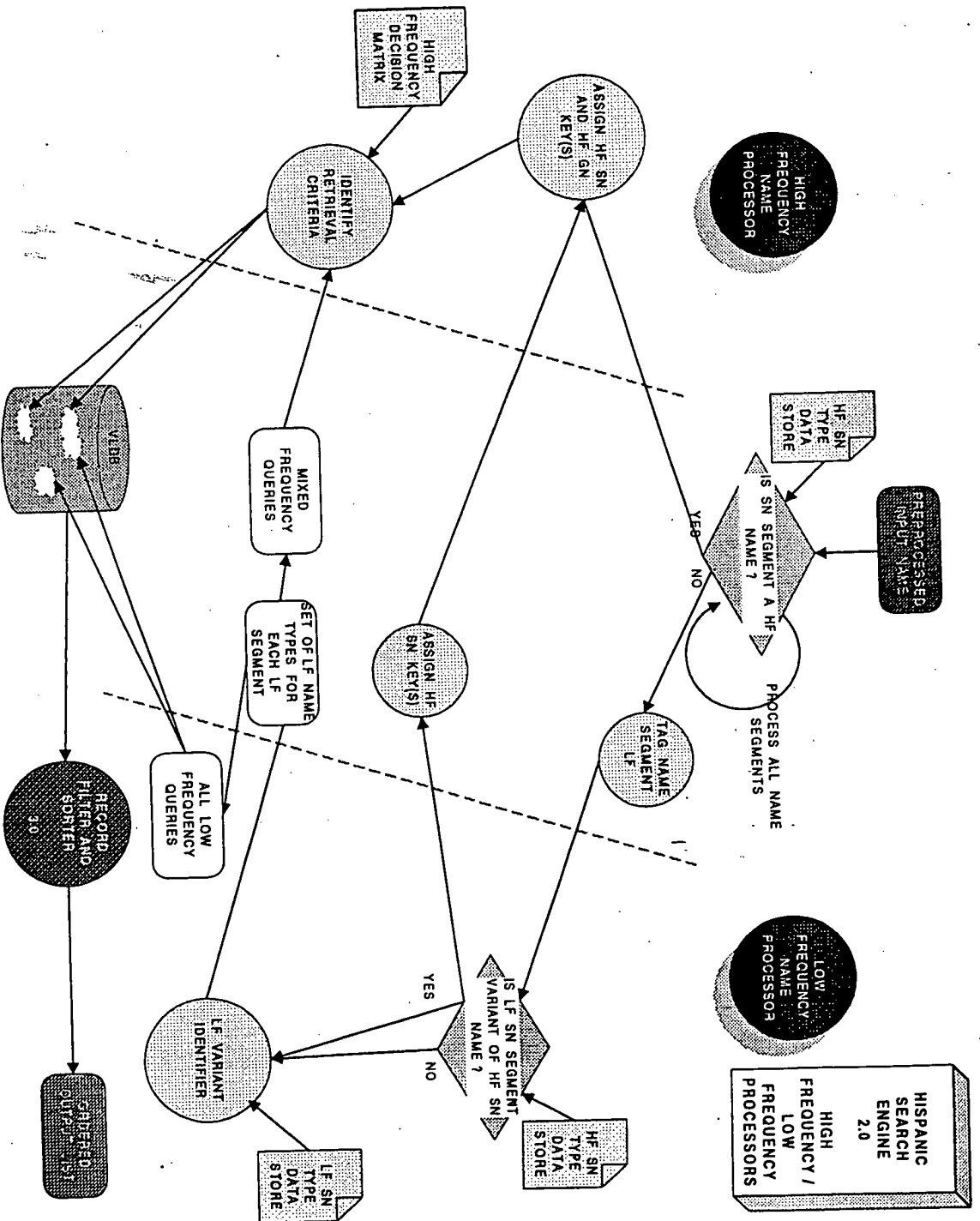
PROCESS FLOW

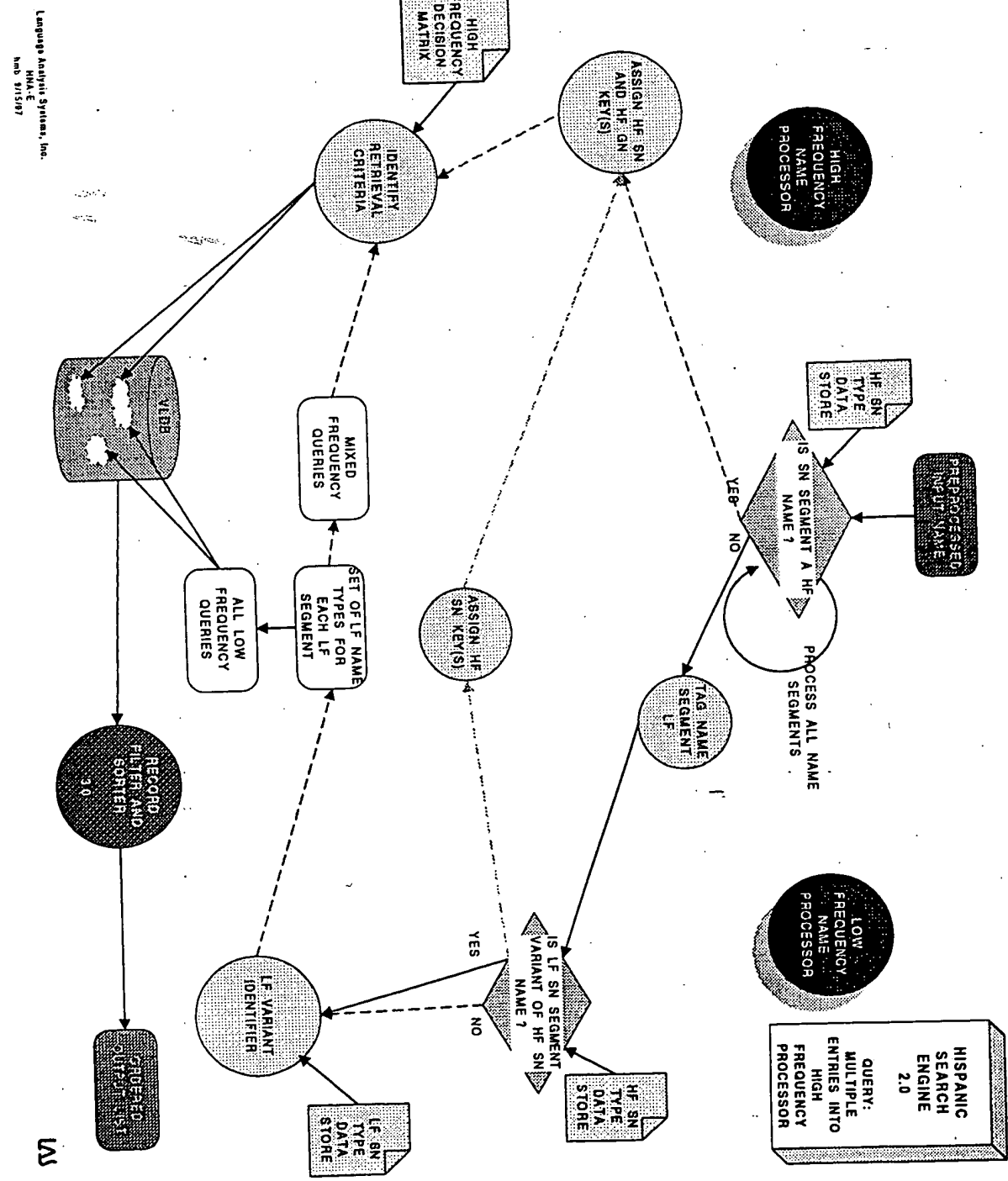




HISPANIC
SEARCH
ENGINE
2.0
OVERVIEW

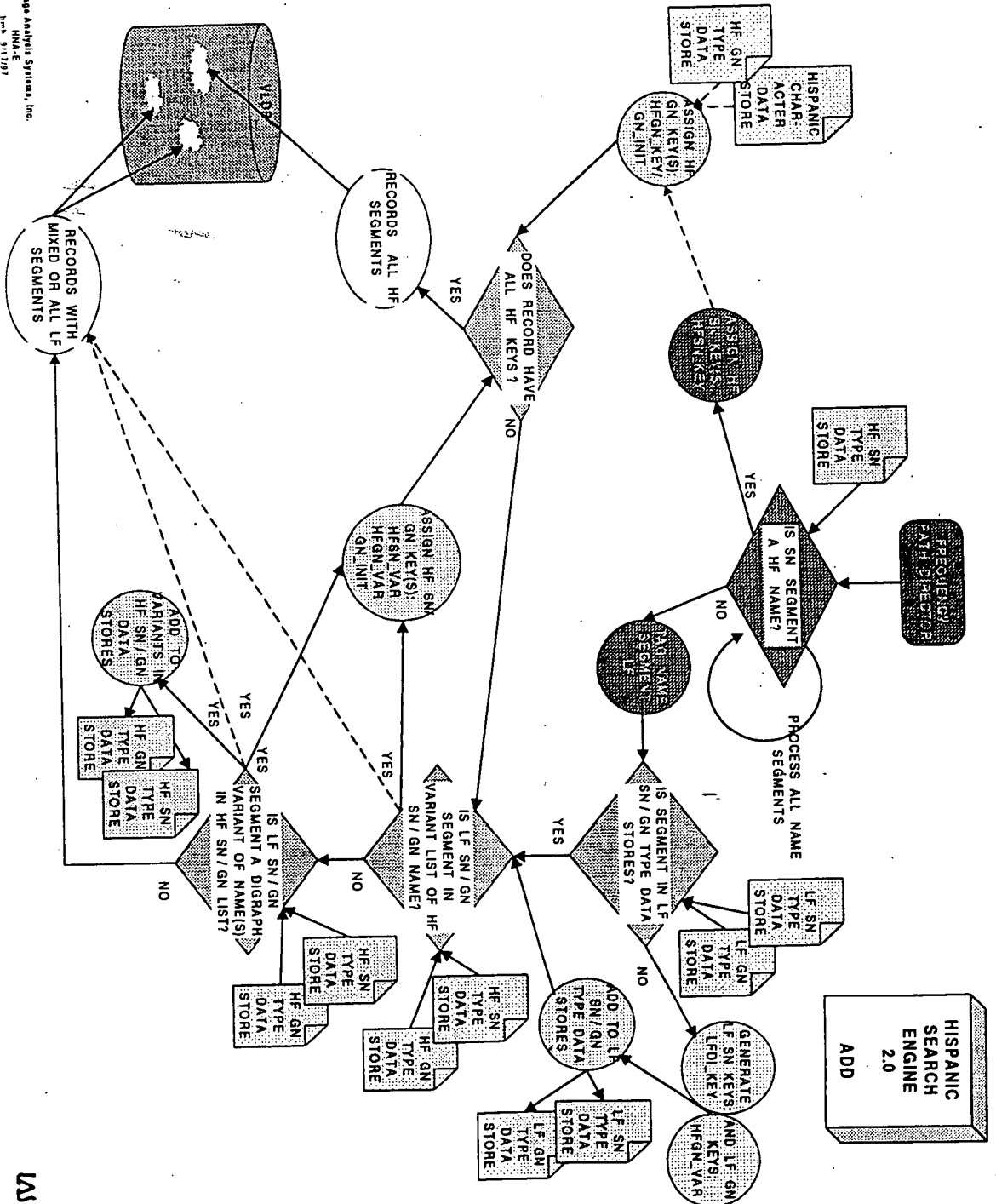


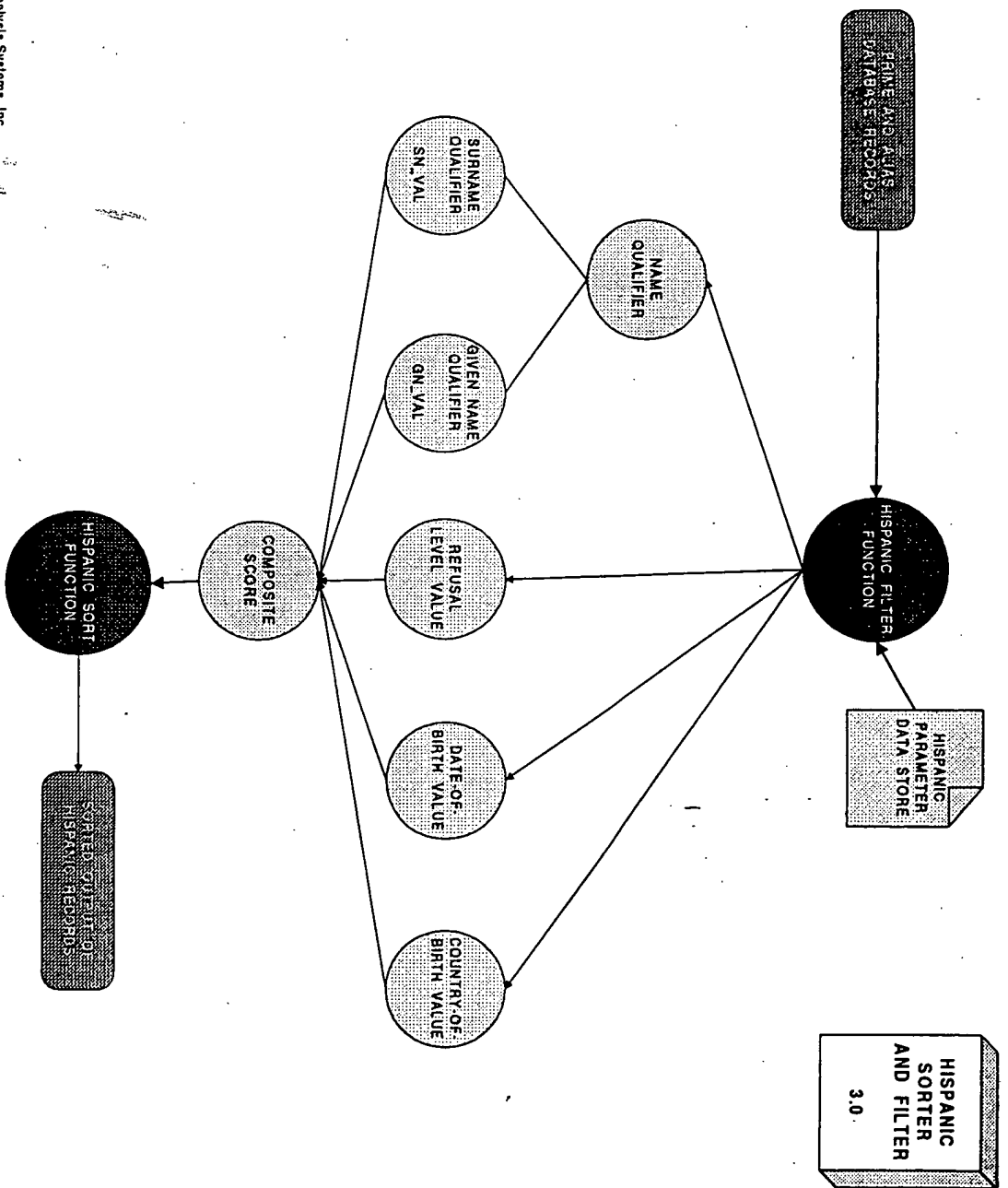




Language Analysis Systems, Inc.
 HMA-E
 Rev 8/15/87

123





HISPANIC NAMESEARCH ALGORITHM FOR CLASS-E (HNA - E)

FREQUENCY PATH DIRECTOR

- FPD directs record based on frequency of SURNAME data only
- ALL surnames must be HFSN_TYPES for record to go directly to HF Processor
- FDP assigns HFSN_KEY (SET_ID in HFST) to each high frequency surname

High Frequency Surname Type (HFST) Data Store (Sample)

ID NO	HFSN_TYPE	SET_ID
0001	GARCIA	0001
0002	RODRIGUEZ	0002
0003	HERNANDEZ	0003
0004	LOPEZ	0004
0005	MARTINEZ	0005
0006	GONZALEZ	0006
0007	PEREZ	0007
0008	SANCHEZ	0008
0009	RAMIREZ	0009
0010	GOMEZ	0010
0011	...	0011

GARCIA LOPEZ, ANTONIO JESUS
0001 0004

HIGH FREQUENCY PROCESSOR

BOMEZ PEREZ, JOSE WILLIAM

0007

LOW FREQUENCY PROCESSOR

HIGH FREQUENCY PROCESSOR

- High Frequency Processor assigns HFGN_KEY (SET_ID in HGT) to each High Frequency Given Name

Hispanic Given Name Type (HGT) Data Store (Sample)

ID NO	GN TYPE	SET ID	HI FREQ	GND
0001	JOSE	0001	1	M
0002	MARIA	0002	1	F
0003	JUAN	0003	1	M
0004	LUIS	0004	1	M
0005	ANTONIO	0005	1	M
0006	CARLOS	0006	1	M
0007	JESUS	0007	1	M
0008	MANUEL	0008	1	M
0009	FRANCISCO	0009	1	M
0010	JORGE	0010	1	M
0011	...	0011
2367	DAGOBERTO	0000	0	M

GARCIA LOPEZ, ANTONIO JESUS
0001 0004 0005 0007

IF ALL NAME SEGMENTS HAVE BEEN ASSIGNED HFSN_KEYS AND HFGN_KEYS, THE HFP MATRIX ACCESSES
THE HISPANIC DECISION MATRIX

- HFP accesses Hispanic Decision Matrix for additional search criteria

Hispanic Decision Matrix (HDM) (Sample)												
Single-Segment SN				Two-Segment SN								
QUERY SN FORMAT	A	A	A	AB	AB	AB	AB	AB	AB	AB	AB	AB
DATABASE SN FORMATS	A	AB	BA	A	B	AC	CA	CB	BC			
YR#	5	5	2	5	4	4	2	0	0	0	0	0
RL#	4	4	3	4	4	4	1	0	0	0	0	0
RGNDR	MFU	MFU	MFU	MFU	MFU	MFU	MFU	MFU	MFU	MFU	MFU	MFU

GARCIA LOPEZ, ANTONIO JESUS
0001 0004 0005 0007

For example, records with following surnames and retrieval criteria would be retrieved:

NAME	YR#	RL#	RGNDR
GARCIA LOPEZ	5	4	MFU
LOPEZ GARCIA	4	4	MFU
GARCIA	4	4	MFU
LOPEZ	4	4	MFU
GARCIA MARTIN*	2	1	FU
MARTIN* GARCIA	2	1	MFU
MARTIN* LOPEZ	0	0	MFU
LOPEZ MARTIN*	0	0	MFU

* Any SN segment

SEND TO HISPANIC SEARCH ENGINE

- All HFSN_KEYS
- All HFGN_KEYS
- All Search Criteria

*HISPANIC SEARCH ENGINE WILL RETRIEVE

- AN EXACT MATCH AND
- ALL RECORDS WITH SN KEYS WITH RETRIEVAL CRITERIA AND AT LEAST ONE HFGN_KEY

LOW FREQUENCY PROCESSOR

(1) High Frequency Access

- LFP determines if LF SN is variant of HF SN
- LFP assigns HFSN_VAR keys (ID_NO in HFSV) to SN that is variant of High Frequency Surname

High Frequency Surname Variant (HFSV) Data Store (Sample)

ID_NO	HFSN_VAR	SET_ID	DI_VAL
032711	PEREZ	0007	1.00
032712	PEREZ	0007	0.67
032713	PEREZA	0007	0.77
016976	GOMEZ	0010	1.00
016977	GOMEZ	0010	0.67
016978	BOMEZ	0010	0.67

BOMEZ PEREZ, JOSE DAGOBERTO
016978 0007

- LFP SENDS NAME WITH ALL HFSN_KEYS and HFSN_VAR KEYS TO HFP
- HFP WILL GENERATE GIVEN NAME KEYS
 - HFP WILL IDENTIFY SEARCH CRITERIA IN HISPANIC DECISION MATRIX

(ADDITIONAL) GIVEN NAME KEYS

Hispanic Given Name Type (HGT) Data Store (Sample)				
ID_NO	GN_TYPE	SET_ID	HI_FREQ	GNDR
0001	JOSE	0001	1	M
0002	MARIA	0002	1	F
0003	JUAN	0003	1	M
0004	LUIS	0004	1	M
0005	ANTONIO	0005	1	M
0006	CARLOS	0006	1	M
0007	JESUS	0007	1	M
0008	MANUEL	0008	1	M
0009	FRANCISCO	0009	1	M
0010	JORGE	0010	1	M
0011	...	0011
2367	DAGOBERTO	0000	0	M

EXAMPLE 1:

BOMEZ PEREZ, JOSE DAGOBERTO
016978 0007 0001 D

- JOSE WILL BE ASSIGNED HFGN_KEY
- DAGOBERTO (WHILE IN THE LIST OF GIVEN NAMES) IS A LOW FREQUENCY GN
- DAGOBERTO IS ASSIGNED A GN_INIT KEY OF D

EXAMPLE 2:

Hispanic Character (HCD) Data Store (Sample)

SET ID	CHAR	CHAR VAR
001	B	B
001	B	V
002	S	S
002	S	Z
004	C	C
004	C	S
...		
037	F	F
052	K	K
078	M	M
078	M	N
...		
093	J	J
093	J	H

BOMEZ PEREZ, JOSSE DAGOBERTO
 016978 0007 ----
 (093) J/H (005) D

- BOTH JOSSE AND DAGOBERTO ARE LOW FREQUENCY GIVEN NAMES
- BOTH JOSSE AND DAGOBERTO ARE ASSIGNED A GN_INIT KEY (093 (J/H) AND 005 (D))
 - INITIAL VARIANTS ARE ACCESSED IN THE HISPANIC CHARACTER DATA STORE (HCD)

EXAMPLE OF RETRIEVAL WITH HIGH FREQUENCY SURNAME KEYS AND MIXED GN KEYS

Represents partial set of query patterns only (SLOPEZ will generate additional LF Keys)

QUERY #1	RODRIGUEZ	SLOPEZ	JOSSE	CARLOS	CRITERIA
HFSN_KEY	002				
HFSN_VAR Key		00976			
HFGN_KEY			041 (J, H)	0007	
GN_INIT Key(s)					
HDM FORMATS:					
1	RODRIGUEZ (002)	LOPEZ (000976)			YOB5, RL4, MFU, GN initial = J or H; or GN= 0007
2	LOPEZ	RODRIGUEZ			YOB4, RL4, MFU, GN initial = J or H; or GN= 0007
3	RODRIGUEZ				YOB4, RL4, MFU, GN initial = J or H; or GN= 0007
4	LOPEZ				YOB2, RL1, MFU, GN initial = J or H; or GN= 0007
5	RODRIGUEZ	*			YOB2, RL1, FU, GN initial = J or H; or GN= 0007
6	LOPEZ	*			YOB0, RL0, MFU, GN initial = J or H; or GN= 0007
7	*	RODRIGUEZ			YOB0, RL0, MFU, GN initial = J or H; or GN= 0007
8	*	LOPEZ			YOB0, RL0, MFU, GN initial = J or H; or GN= 0007

LOW FREQUENCY PROCESSOR

(2) Low Frequency Surname Processor

(Includes all SN *not* identified as HF, even HF Variant SN)

Low Frequency Surname Type (LFST) Data Store (Sample)

ID NO	LFST TYPE	LFST KEY
000001	AALVAREZ	AA1
000001	AALVAREZ	AA2
000001	AALVAREZ	AL2
000001	AALVAREZ	AL1
000001	AALVAREZ	AL3
000001	AALVAREZ	LV3
000001	AALVAREZ	LV2
000001	AALVAREZ	LV4
000001	AALVAREZ	VA4
000001	AALVAREZ	VA3
000098	BARRIOS	BA1
000098	BARRIOS	BA2
...		

BOMEZ PEREZ, JOSSE DAGOBERTO

016978 0007

(093) J/H (005) D

- (IN ADDITION TO TREATMENT AS POSSIBLE VARIANT OF HF SN)
- ALL SN *NOT* IDENTIFIED AS HF SN WILL UNDERGO PROCESSING AS LOW FREQUENCY SURNAME

WHAT IS SIMILAR?

HIGH FREQUENCY SURNAMES

QUERY	DATABASE
GARCIA	BARCIA
GARCIA	GARICA

GOMEZ	GAMEZ
GOMEZ	BOMEZ

RAMIREZ	RAMIRES
RAMIREZ	AMIREZ

LOW FREQUENCY SURNAMES

QUERY	DATABASE
BARCIA	GARCIA
BARCIA	*BARCA

BOMEZ	GOMEZ
BOMEZ	*BROMEZ

AMIREZ	RAMIREZ
AMIREZ	*AMIRO
AMIREZ	*ARAMEZ

*RELATED TO LOW FREQUENCY NAME BUT NOT TO HIGH FREQUENCY NAME

LOW FREQUENCY PROCESSOR

(2) Low Frequency Surname Processor (cont.)

BOMEZ

HIGH FREQUENCY RELATIONSHIPS: (may be several relationships)	HFSN_VAR KEYS	GOMEZ
LOW FREQUENCY RELATIONSHIPS:	LFDIKEYs (Base and Positional)	BO1 OM2 ME3 EZ4 BO2 OM1 OM3 ME2 ME4 EZ3
	LFDIKEY Threshold	AMEZ, BOEZ, BOM, BOMEZ, BOMERO, OMEZA, SGOMEZ, THOME....
	DIGRAPH COMPARISON (LF_DI Threshold) (Note: GOMEZ retrieved with HFSN_VAR Key)	BOEZ, BOM, BOMEZ
	DI_KEY	013025 (BOEZ), 013454 (BOM), 013465 (BOMEZ)

- EACH DI_KEY USED AS EXACT MATCH KEY
- NAME MAY HAVE MULTIPLE HFSN_VAR KEYS

SAMPLE OF RETRIEVAL WITH LOW FREQUENCY SURNAMES

Example of Query Formats with Mixed Frequency Surnames

QUERY #1	THORET	SLOPEZ	JOSSE	CARLOS	CRITERIA
HFSN KEY					
HFSN VAR Key		00976			
DI KEY	000632 (THORET)				
	000714 (TOREAT)			0007	
HFGN KEY			041 (J, H)		
GN INIT Key(s)					
HDM FORMATS:					
1	THORET	LOPEZ (000976)			YOB3, RL4, MFU, GN initial = J or H; or GN= 0007
2	LOPEZ	THORET			YOB4, RL4, MFU, GN initial = J or H; or GN= 0007
3	THORET				YOB4, RL4, MFU, GN initial = J or H; or GN= 0007
4	LOPEZ				YOB2, RL1, MFU, GN initial = J or H; or GN= 0007
5	THORET	*			YOB2, RL1, FU, GN initial = J or H; or GN= 0007
6	LOPEZ	*			YOB0, RL0, MFU, GN initial = J or H; or GN= 0007
7	*	THORET			YOB0, RL0, MFU, GN initial = J or H; or GN= 0007
8	*	LOPEZ			YOB0, RL0, MFU, GN initial = J or H; or GN= 0007
1	TOREAT	LOPEZ			YOB3, RL4, MFU, GN initial = J or H; or GN= 0007
2	LOPEZ	TOREAT			YOB4, RL4, MFU, GN initial = J or H; or GN= 0007
3	TOREAT				YOB2, RL1, MFU, GN initial = J or H; or GN= 0007
4	LOPEZ				YOB2, RL1, FU, GN initial = J or H; or GN= 0007
5	TOREAT	*			YOB0, RL0, MFU, GN initial = J or H; or GN= 0007
6	LOPEZ	*			YOB0, RL0, MFU, GN initial = J or H; or GN= 0007
7	*	TOREAT			YOB0, RL0, MFU, GN initial = J or H; or GN= 0007
8	*	LOPEZ			YOB0, RL0, MFU, GN initial = J or H; or GN= 0007

DATABASE RETRIEVAL

HIGH FREQUENCY RECORDS

(about 50% of the Hispanic data contain only HF segments; well over 55% contain HF SN with any type GN)

- USE SEARCH CRITERIA FROM HISPANIC DECISION MATRIX
 - ONE OF HFGN_KEYS MUST MATCH
- RESULT = RECORDS WITH HIGH FREQUENCY SURNAMES AND PRE-DETERMINED SURNAME VARIANTS ACCORDING TO SEARCH CRITERIA LIMITED BY GIVEN NAME AND RECORD GENDER

MIXED FREQUENCY RECORDS

(over 26% of Hispanic data contain mixed HF and LF surname segments)

- USE SEARCH CRITERIA FROM HISPANIC DECISION MATRIX FOR LF SN THAT ARE HF VARIANTS
 - DETERMINE LIST OF VARIANTS FOR LF SURNAME AND GENERATE ADDITIONAL QUERIES
 - ONE OF GN KEYS MUST MATCH
- RESULT = RECORDS WITH HIGH FREQUENCY SURNAME (AND VARIANTS), HIGH FREQUENCY SURNAME RELATED TO LF SURNAME AND LF SURNAME VARIANTS; SOME LIMITATION BY SEARCH CRITERIA, GIVEN NAME AND RECORD GENDER

LOW FREQUENCY RECORDS

(about 15% of Hispanic data contain only LF surname segments)

- EXACT MATCH ON BOTH LF SN VARIANTS IN EITHER POSITION OR ALONE WITH RLYOB RESTRICTION
 - ONE LF SURNAME VARIANT IN EITHER POSITION WITH YOB = EXACT YOB AND RL = 00 OR TYPE 1 SERIOUS

HISPANIC FILTER AND SORTER

(1) EXACT MATCH

(2) SCORES

SN_VAL
GN_VAL

(3) PARAMETERS

SNTHR
GNTHR
ASVAL
AGVAL
OPVAL
OPGVAL
INTSN
INITGN
TAQASN
TAQAGN
TAQXSN
TAQXGN
RGNDR

| Database Records Retrieved with HFSN_KEYs (Sample)

QUERY	SN#1	HFSN_KEY	DI_VAL	SN#2	HFSN_KEY	DI_VAL
	GARCIA	0001		GOMEZ	0010	
DATABASE RECORDS	GARCIA	0001	1.00	BOMEZ	0010	0.67
	BARCIA	0001	0.71	GAMEZ	0010	0.67
	LOPEZ	0004	0.17	GARCIA	0001	1.00

RECORD EVALUATION

Example of Surname Evaluation

SN Parameter Evaluation: OPSN Applies

	GARCIA	GOMEZ
BOMEZ		
GARCIA	$1.00 * 0.65 = 0.65$	$0.67 * 0.65 = 0.44$

SN Parameter Evaluation: ASVAL Applies

	GARCIA	GOMEZ
GARZA	0.62	
GOMEZ		$1.00 * 0.65 = 0.65$

- DETERMINE SN_VAL
- DETERMINE GN_VAL
- DETERMINE COMPOSITE SCORE:

$SN_VAL * GN_VAL * RL\# PARM_VAL * YOB\# PARM_VAL * COB\# PARM_VAL$

- ORDER RECORDS
(1) EXACT MATCH
(2) BY COMPOSITE SCORE

The Use of Phonological Information in Automatic Name Searching

Richard Lutz, Ph.D.
AIPA97

March 25, 1997

**PROPRIETARY
INFORMATION**



© Language Analysis Systems, Inc.
2214 Rock Hill Road—Herndon, VA—20170

AIPA97 Paper Presentation:
The Use of Phonological Information in Automatic Name Searching

Richard Lutz, Ph.D.

© Language Analysis Systems, Inc.
at the Center for Innovative Technology
2214 Rock Hill Road -- Herndon, VA 20170

Application Area: Automated Data Understanding

Abstract

This paper describes a two-year research effort to incorporate phonological information into automated name searching. Specifically, names represented by standard roman characters are automatically converted to multiple phonetic representations, based on sets of regular expressions that relate character strings to predictable sounds or sound sequences using a widely accepted phonetic notation system, the International Phonetic Alphabet. Names are retrieved when there is an intersection of the regular expression of the query name with regular expressions of names in a preprocessed database. Additional similar names can be retrieved based on the articulatory characteristics of the sound segments contained in the query and database names.

1.0 Introduction

Variation in the spellings of names is a persistent issue in the area of automated name searching in large databases (Hermansen, 1985). In general, the source of spelling variation of names can be analyzed and explained a posteriori. Predicting any individual spelling, however, remains problematic. Sources for spelling variation include: ~~keyboard-~~ **based data entry errors (e.g., hitting the wrong key: Genning for Henning), syntactic variation (e.g., out-of-sequence given name and surname such as Richard Thomas for Thomas Richard), morphological variation (e.g., truncated strings such as Rich or R for Richard) and semantically-based variation (e.g., nativizations such as Goldwater for Goldwasser).** Of interest in the current paper is variation due to orthographic conventions (e.g., English can represent the same sound in more than one way, as in Stephen ~ Steven) and articulatory variation (e.g., the *p* in Thompson is a predictable spelling of Thomson based on principles of articulation). While there are multiple sources of name variation, this paper will present evidence 1) that the inherent ambiguity in the English use of roman characters can be mitigated by multiple mappings to unambiguous phonetic characters and 2) that phonologically-similar names can be retrieved through the analysis of sounds into their articulatory features (i.e., place and manner of articulation). It is based on research conducted from September of 1995 through the present.

2.0 Statement of Problem

Character-based name searching relies on spelling as the basis for calculating distance between the query name and the database name. While spelling using roman characters is not unrelated to pronunciation, the relationship between the two is often inconsistent (Cummings 1988), and the orthographic information (i.e., conventions of the spelling system of a language) is at times misleading. Thus, one spelling may map to multiple pronunciations: *Lutz* can be pronounced to rhyme with *puts*, *cuts* or *shoots*, and at least several additional non-English pronunciations are possible. The converse, of course, is also the case: there may be a number of ways of representing a single pronunciation: *Lewis* and *Louis*, for example, are usually pronounced identically by English speakers.

Character-matching techniques assume a reliable relationship between the orthographic system and the pronunciation. This assumption is flawed because the *goodness of fit* between orthography and pronunciation, especially for English, is *many-to-many*, that is, a given roman character can stand for more than one sound, and an individual sound may be represented in more than one way in the spelling system. Thus, the sound [f] can be written as *f* (*Frank*), *ff* (*Taffy*), *ph* (*Phillip*) or even *gh* (*Rough*). Conversely, the *gh* digraph may represent the [f] sound of *Rough*, be silent (*Dough*), or represent [k] (in some pronunciations of *McCloughlin*), [h] (in *Monaghun*), [g] (in *McGhee*) or [gh] (across syllable breaks, as in *Bighouse*).

While much name variation can be traced to non-phonological issues, including syntax (order of name segments), aliases (*John Doe* for *John Dillinger*), morphological issues (*Peg* for *Margaret*) or data entry errors, many name variants can be traced to the relationship between orthography and pronunciation. Orally transmitted names, for instance, are especially prone to guesses on the part of the transcriber as to the "official" (i.e., legal) spelling of an individual's name. Language contact can account for some spelling variants as well (French *Beauchamp* and Anglicized *Beecham*), as can transcription from non-roman character sets (*Wachmi* and *Quakhmi*, *Xie*, *Hsieh* and *Sye*) and sound change over time (e.g., *Leigh* is now pronounced the same as *Lee*).

Additionally, regular (i.e., predictable) processes of speech produce variability in how a name may be written. Thus, the presence of the letter *p* in *Thompson* is an artifact of poor articulatory timing as the articulators move from a nasal [m] to an oral [s]. (The variant spelling *Thomson* reflects a more etymologically justified spelling.)

3.0 Name Representation: Spelling

LAS has been investigating the feasibility and utility of incorporating information about the pronunciation of characters into the automated name searching process. The researchers considered a number of options, including an acoustic-level of representation and character-based rules, and determined that searching of character-based databases could be enhanced to include predictable language-based information about character-to-

¹ Square brackets indicate that a sound is being represented, rather than a spelling.

sound mappings. Specifically, LAS recommended the use of the stock of phonetic symbols known as the International Phonetic Alphabet (IPA), widely used by linguists to represent the inventory of sounds used in the world's languages, and officially adopted by the International Phonetic Association (Laver, 1994). The IPA uses a closed set of symbols to transcribe speech in ways that are interpretable unambiguously by linguists, regardless of the language being described. (See Appendix A.) For example, the symbol [↓] (placed between brackets to indicate that it represents a sound rather than a letter) always stands for a *voiceless* labiodental fricative, as in English *thigh*, while [↓] always stands for the equivalent *voiced* labiodental fricative, as in English *thy*. Thus, IPA disambiguates the English orthographic pattern of using *th* to stand for either sound: *thigh* [↓aj] versus *thy* [aj]. A name such as *Gaither*, of course, might be pronounced with either of these sounds, and would thus have two IPA representations, one for each pronunciation: [ge↓r] versus [ger]. There is international agreement by members of the International Phonetic Association, founded in 1889, as to the interpretation of IPA symbols. A re-evaluation of the stock of symbols and special diacritic marks took place at the 1989 IPA Convention in Kiel, and the efforts of the Association have resulted in the unambiguous mapping of sounds onto IPA symbols that transcends individual speakers or languages (Laver, *ibid.*).

4.0 Mapping Spelling to Sound

The issue of how to predict pronunciation of names from orthography is far from trivial. Two key considerations include that:

- pronunciations of proper names are far less uniform than pronunciations of other vocabulary. The pronunciation of the noun *dough* is more-or-less fixed in English, despite the fossilized spelling that can be traced to an earlier pronunciation. The pronunciation of the name *Lough* is far less certain: individuals named *Lough* may well vary in their pronunciation of the family name and, even if all families named *Lough* could reach a consensus, there is no assurance that those unfamiliar with their consensus would guess that pronunciation. Additionally, some names retain old spellings that map to modern pronunciations in highly improbable ways (e.g., British *Cholmondeley* is commonly pronounced the same as *Chumley*). Claims of "correct" pronunciations carry little weight in terms of name searching; and:
- orthographies are language-specific. The pronunciation of the letter *x* regularly maps to [ks] and [z] in English (*Alexander*, *Xenia*), is regularly silent word-finally in French orthography (*LaCroix*), stands for the velar fricative [x], or [s] in Spanish (*México*, *Xochimilco*), and a [dz] or [↓] in Albanian (*Hoxha*). Additionally, standardized transcription systems from non-roman systems to roman exploit the letter *x* to stand for other, non-English sounds (e.g., Chinese *Xie*, Greek *Xristos*). Finally, any name may be nativized to fit the "borrower" language: spellings of non-Anglo names may be pronounced according to English orthographic conventions (e.g., French *Duquesne* pronounced [dukwni].)

5.0 Writing IPA Conversion Rules

IPA is an effective notational system for representing pronunciation. LAS has written sets of rules that relate spellings to sounds. The rules are language-based, with sets of rules operating for Arabic, Mandarin Chinese, Hispanic and Anglo names. The rules assume:

- 26-character sets of roman letters, absent all diacritic markings, including accent marks or tone indicators;
- English speakers, either naïve or expert in the language of origin;
- one spelling can map to multiple pronunciations.

The rule sets were written to specific development databases made of single name elements, either surname or given, and taken from a variety of sources, including the U.S. Census list of the most frequent names in the U.S. and large U.S. databases of names from other countries. The names were manually tagged as "Arabic", "Mandarin Chinese", "Hispanic" and "Anglo", where "Anglo" was loosely interpreted to include Western European Germanic names (including Dutch and German). A team of linguists used a variety of sources to determine possible pronunciations, including native speaker knowledge and textual information (e.g., Cummings, 1988, Hanks and Hodges, 1989, 1990, Symonds, 1986). In general, rules were written broadly in order to ensure that most plausible pronunciations were captured. The Arabic and Mandarin Chinese rules included transcription variation (e.g., Chinese pinyin, Wade-Giles and Yale conventions of rendering Chinese names into roman script, as in *Xie/Hsieh/Sye*). The sample Anglo rule below is interpreted to mean that the letters *sc* preceded by anything and followed by the letters *le* can be pronounced as [s] or [sk] (e.g., *Muscle* and *Mosclin*):

sc/ anything __ le → [sk?]

Rules were implemented using standard regular expression notation. The following table shows a sample query and the names returned from a data file containing the 88,799 most frequent surnames from the U.S. census:

<i>Search on SMITH</i>	
	<i>SMITH</i>
	<i>SMYTH</i>
	<i>SMITHE</i>
	<i>SMIT</i>
	<i>SMYTHE</i>
	<i>SMIDT</i>
	<i>SMIHT</i>
	<i>SZMIDT</i>

Figure 1 Search on name *SMITH*

As an example of the advantages of matching on IPA, consider a query on the name *Lee*. Converted to the IPA string [li], exact matches with numerous spelling variants are automatic, including *Leigh* and *Li*. Typical character-based matches will fail to retrieve *Leigh* or *Li*, since the percentage of character overlap is minimal. Conversely, a standard index matching system such as Soundex will categorize *Lee* and *Li* identically, but will still miss *Leigh*, given the presence of a salient letter (g), and will retrieve a large number of names of low relevance, including *Lu*, *Liao*, *Low*, *Louie*, *Luhoya* and *Lehew*.

6.0 Phonological Processes

In addition to predictable spelling variation, rules were written to account for predictable articulatory processes (MacKay, 1987; Wolfram and Johnson, 1982). For example, the variant spellings of *Thomson* ~ *Thompson*, *Simson* ~ *Simpson*, *Demsey* ~ *Dempsey*, etc. can be accounted for by regular movement of the velum (i.e., the soft palate) from a bilabial nasal [m] to an oral [s]. Production of an intrusive bilabial oral [p] is entirely a result of the timing of the movement from nasal to oral articulation. LAS incorporated likely articulatory variation into the IPA rule sets. Thus, a query of the name *Thomson* will retrieve the variant *Thompson* as an exact match.

7.0 Testing the Rule Sets

To test the net effect of the Orthography-to-IPA rules, LAS conducted a controlled test of the rules by randomly selecting 157 test names from a database of 55,545. The database contained names that were from sources identified as Arabic, Mandarin Chinese, Hispanic and Anglo (again, broadly defined). A native speaker of educated standard American English was asked to record the 157 test names using pronunciations of his choosing. The audio recordings were played for native speakers of American English, who were asked to write one or more "likely" spelling for each name. LAS elicited 3,689 variants in all by playing the recordings to native speakers of American English. The variant spellings were then used as test query names to calculate the retrieval rates of the original name spellings. Overall, 69% of all variant spellings were retrieved by the IPA rules. However, qualitative analysis of the results showed that approximately 23% of the variant names not retrieved were due to perceptual mishearings of the recorded names. For example, the variant spellings of the test name *Baughn* predictably included *Bahn*, *Baun*, and *Bonn*, and the IPA Conversion Rules succeeded in mapping all to the original test name spelling. However, a fourth elicited spelling, *Vaughn*, was not predicted, and the IPA Conversion Rules did not map it to *Baughn*. The mishearing of [v] for [b] is not unusual, given the acoustics shared by the two sounds. The IPA Conversion Rules, which include regular articulatory variants such as *Thomson/Thompson*, were purposely not intended to retrieve perceptually similar names during the current phase of research.

8.0 Fuzzy Matches: Articulatory Similarity

At the heart of the research has been an effort to improve the automatic name searching process by retrieving names that are *similar* to the query name. The IPA Conversion Rules are able to capture a good deal of name variation that can be attributed to orthographic sources, whether intralingual (e.g., *Leigh/Lee*) or interlingual (e.g., transcriptions to roman orthography from Chinese: *Xie ~ Hsieh ~ Sie*). An additional goal has been to retrieve names that are not phonologically identical to the query name, but that a careful analyst would like to consider before abandoning a search. Thus, while spelling variants of the name *Benke* include *Behnke* and *Benck*, the analyst might want to consider names that seem phonologically close to the query name without being a predictable variant (e.g., *Benge*, *Bunkey* and perhaps even names like *Penke*, *Panke* or *Benische*). While most search algorithms permit fuzzy matches, these are invariably based on calculations of number of characters shared. From the perspective of character matching, the letter *b* is as different from the *p* as it is from *x*, *y* or *z*. Thus, to permit retrieval of *Penke* for *Benke* is to require retrieval of any name that differs from the query by the first character, including *Xenke*, *Yenke* and *Zenke*. This clearly does not follow any phonologically reliable principle, and significantly reduces the efficiency of automatic retrieval. Even indexed systems, such as Soundex, group letters as either co-indexed or unrelated. Thus, while Soundex is often called "phonetic" because it groups letters that share some phonological characteristics, it cannot compare the degree to which two sounds, or indeed two names are related: it lacks granularity. Thus, Soundex would treat *Benke*, *Penke* and *Panke* as identical rather than similar. Soundex would exclude *Benische* from the group because of the letter *i* in the spelling, in effect treating *Benische* as being equally distant from *Benke* as from *Smith*.

It is clear, however, that sound segments can be analyzed in terms of their articulatory characteristics, and that some sounds fall into natural categories, such as vowels and consonants. Properties of sounds have been described in detail by a number of linguistic analyses according to place and manner of articulation (e.g., [p] and [b] are both articulated at the lips by complete blockage of the air flow and sudden release of pressure). One of the best known descriptions of phonetic classification is that of the American linguists Chomsky and Halle (1968). All the distinct sounds of American English can be described using 15 distinctive features (see Appendices B and C). By classifying sounds according to these distinctive features, a fairly clear picture emerges of how close any two sounds are to one another. Thus, [p] and [b] differ by just one feature, voicing, while [p] and [f] differ by three and [p] and [v] by four. In general, articulatory distance can be counted in terms of how many articulatory characteristics sounds share.

LAS created a file of feature differences between pairs of sounds, essentially mapping phonetic features onto IPA notation. By relaxing the threshold of allowable differences, increasingly distant sounds are retrieved. Thus, by permitting matches of IPA characters that are not exact matches, names are retrieved that are phonologically close. Even IPA sound-to-sound comparisons yield interesting sets of names for comparison. By relaxing

retrievals to include single feature differences, a search of the name *Smith* now brings back these additional names:

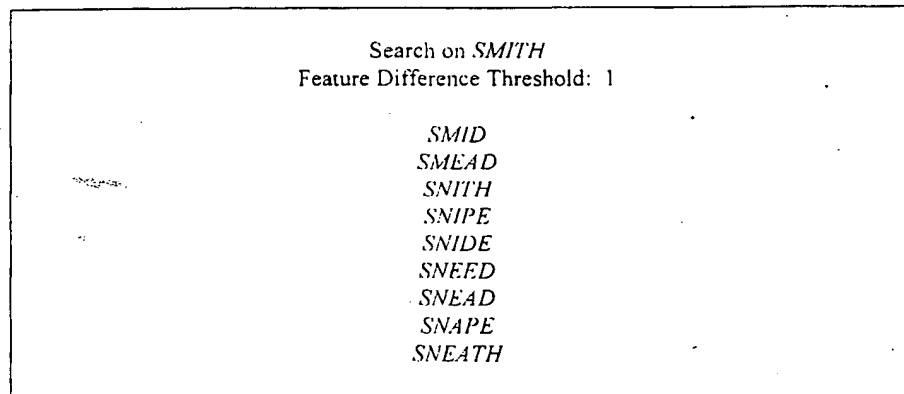


Figure 2 Fuzzy Search on *Smith* measuring Phonetic Feature Differences

Viewed in physiological terms, this is reasonable. Phonetic features refer to salient characteristics of articulation, so that differences generally reflect how likely it is that any two sounds would be articulated in place of another. There are numerous additional factors, of course, that ought to be considered in measuring how similar two names are to one another articulatorily.

9.0 Final Sorting of Names Retrieved

The names retrieved by searches on phonetic features may not all be of equal relevance to the query name. Additional factors are under consideration to sort names retrieved, based on a variety of phonological characteristics.

9.1 Sonority Level

The differences in phonetic features generally express the amount of effort needed to move articulators from one sound to another. The sounds [p], [t] and [k] form a natural class of voiceless stop consonants — identical in manner of articulation. All are extremely common in the world's languages, and are among the first acquired by children. They differ in place of articulation, and this is reflected in feature differences. However, manner of articulation is probably a better measure of energy expenditure than is place of articulation: voiceless stops are all extremely low in sonority, that is, the amount of energy needed to produce a sound. Vowels, on the other hand, require much more effort: they, in essence carry the sound wave. In order for feature differences to effectively measure level of effort required, differences should be weighted according to sonority level. In general terms, sounds fall into nine levels of sonority, with voiceless stops [p], [t] and [k] at the

low end and the vowels [] as in *father* and [ə] as in *fun* at the most sonorous end (Ladefoged, 1982). Sorts of names retrieved ought to consider the sonority value of sounds. This might be accomplished by weighting phonetic features or by a more complicated comparison of sonority level contours of names or syllables.

9.2 Syllabification

Additionally, in languages that time segments based in part on stress patterns, it is reasonable to compare stressed syllables to one another. In the following example, names have been aligned in terms of substrings, in this case corresponding to syllables:

Chester: [tʃ σtp]
Chesterton: [tʃ σtp tv]
Winchester: [w v tʃ σtp]

Both in terms of articulatory effort (sonority) and psychological salience, it would be misleading to treat all three occurrences of the substring [tʃ] as equivalent: stress clearly must be included in the equation. LAS has written a syllabifier that automatically parses English IPA strings, including names, according to a set of rules. Future research will investigate the possibility of ranking similar names through analysis at the syllabic level. Syllabic level analysis has the strength of lining up comparable substructures of names. All syllables share the same internal structures (i.e., onset of the syllable, nucleus, and coda), and alignment by syllable enables meaningful comparisons of internal structures of names (where a period represents the syllable break):

Linda [λ v . δ]
Lisa [λ ɪ _ . σ]

Note that in the above example, the coda (i.e., end) of the first syllable in *Linda* is filled by [n] but empty in *Lisa*, as indicated by the underscore. A meaningful comparison of the two names would compare the [n] of *Linda* to an empty coda rather than to the [s] in the onset (i.e., beginning) of the second syllable of *Lisa*.

9.3 Position in Name

Some weight ought to be given to absolute initial position in names. Many indexed systems, including Soundex, key names to the initial letter. This is, of course, problematic, since the initial letter may be silent or part of a digraph (e.g., *Knox*, *Philip*). However, indexing on the first sound, or at least considering the first sound as more significant than sounds in other positions may be warranted. This, like syllable-level comparisons, will probably be a factor in final sorting of names retrieved.

9.4 Non-Phonological Factors in Sorting of Names Retrieved

Certainly, it must be acknowledged that non-phonological levels of analysis may be critical to any useful definition of *similarity*. Morphological units – word parts that may contain

semantic information, including prefixes and suffixes — such as *Mc-*, *-ton*, and *-sky* are likely sources of variations. Thus, *Lubin* and *Lubinsky* are critically related (in terms of their roots), while *Lubin*, *Rubin* and *Lupine* are very close in terms of articulation. The morphological factor could be handled efficiently with a look-up list of morphological elements, but this remains outside the current scope of this project.

Similarly, orthography itself might play a useful role in the final sort of names retrieved. The following names retrieved for a fuzzy search on the name *Bucket* have been sorted using a simple sort on letters.

Search on BUCKET
Feature Difference Threshold: 1

BYXKETT
 BEXKET
 BIXKET
 BYØYET
 BEXKETT
 BIXKETT
 BYXHHEIT
 BOØYET
 BAØYET
 BOXHAT
 BYXHITE
 BEXKQITH
 BAXOT
 BOOKOYT
 BAXOTE
 BOYØYET
 BEKHIT
 BOQXYTT
 BEØYETTE

Figure 3 Search on *Bucket* Sorted by Spelling

Current plans are for a final ranking of names retrieved based on a combination of factors, including number of syllables, stress, weighting of features by sonority levels and name-initial segments.

10.0 Conclusions

In sum, automatic name searching can benefit in three ways from incorporation of phonological information:

- leveling differences due exclusively to orthographic mapping;
- leveling differences due to predictable phonological processes, such as intrusive consonants; and
- retrieving additional names that contain phonologically similar sounds to those of the query name.

Having retrieved phonologically relevant names, a phonologically-enhanced name search engine can then sort names using a multiple factor weighting scheme.

LAS views this technology as extremely promising, offering a tool to enhance current automatic name searching, increasing chances of retrieving name variants that character-based systems miss by retrieving and sorting names in a phonologically principled way.

Appendix A: Descriptions of IPA Symbols

Phonetic symbol	Description	Example
p	voiceless bilabial stop	p in the English name Peter
b	voiced bilabial stop	b in the English name Buddy
ɸ	voiceless bilabial fricative	f in the Japanese name Fujimori
β	voiced bilabial fricative	b in the Spanish word saber
m	bilabial nasal	m in the English name Mary
ɥ	voiced rounded palatal approximant	u in the French name Nuit
f	voiceless labio-dental fricative	f in the English name Fred
v	voiced labio-dental fricative	v in the English name Vera
ɸ̃	voiced labio-dental nasal	n in the Italian word anfora
t	voiceless alveolar stop	t in the English name Ted
d	voiced alveolar stop	d in the English name Doug
θ	voiceless apico-dental fricative	th in the English name Theodore
x	voiced apico-dental fricative	th in the English name Rather
s	voiceless alveolar fricative	s in the English name Sam
z	voiced alveolar fricative	z in the English name Zachary
n	voiced alveolar nasal	n in the English name Nathan
l	voiced alveolar lateral	l in the English name Linda
ɬ	voiceless alveolar lateral fricative	ll in the Welsh name Llewellyn
ɮ	voiced alveolar lateral fricative	dh in the Zulu word dhla (to eat)
ɹ	voiced alveolar continuant	r in the English name Richard
ɾ	voiced apico-alveolar trill	r in the Spanish name Ricardo
ɸ̥	voiced alveolar flap	tt in the English name Ritter
ɸ̥	voiceless retroflex stop	as in the Arabic name Tariq
ɸ̥	voiced retroflex stop	as in the Arabic word difda' (frog)
ɸ̥	voiceless retroflex fricative	as in the Arabic name Sabir
ɸ̥	voiced retroflex fricative	as in the Arabic name Dhafir
ɸ̥	voiced retroflex nasal	Marathi (India)
ɸ̥	voiced retroflex lateral approximant	Marathi (India)
ɸ̥	voiced retroflex flap	d as in Hindi dal (lentil stew)
ɸ̥	voiceless palato-alveolar fricative	sh in the English name Sheila
ɸ̥	voiced palato-alveolar fricative	z in the English word azure
ɸ̥	voiceless alveo-palatal fricative	x as in the Chinese name Xia
ɸ̥	voiced alveo-palatal fricative	ʃ in the Polish word ʃle
ɸ̥	voiceless palato-alveolar affricate	ch in the English name Charlie
ɸ̥	voiced palato-alveolar affricate	j in the English name Jennifer
ɸ̥	voiced palatal nasal	ɲ in the Spanish word Doña
ɸ̥	voiced palatal lateral approximant	ll in the Spanish word calle (street)
k	voiceless velar stop	k in the English name Kim
g	voiced velar stop	g in the English name Gary
x	voiceless velar fricative	x in the Spanish name Jose
ɣ	voiced velar fricative	g in the Spanish word luego (later)
ŋ	voiced velar nasal	ng in the English name Bing

Appendix A: Descriptions of IPA Symbols (Continued)

Phonetic symbol	Description	Example
ɸ	voiceless velar lateral	l in the Polish <i>Walesa</i>
ʋ	voiceless labio-velar approximant	wh as in the English name <i>White</i> (for some speakers)
w	voiced bilabial approximant	w in the English name <i>Wayne</i>
q	voiceless uvular stop	as in the Arabic name <i>Qasim</i>
ʁ	voiced uvular stop	Eskimo and Tehrani Persian
χ	voiceless uvular fricative	ch as in the German word <i>Buch</i>
ʀ	voiced uvular fricative	r in some Parisian pronunciations of the French name <i>RenJe</i>
ɴ	voiced uvular nasal	n in the Eskimo word <i>eNima</i> (melody)
ʀ	voiced uvular trill	r in the French name <i>RenJe</i>
ħ	voiceless pharyngeal fricative	h as in the Arabic name <i>Muhammad</i>
ʕ	voiced pharyngeal fricative	as in the Arabic name <i>Sa'ad</i>
ʔ	voiceless glottal stop	tt as in the English name <i>Sutton</i> or the word <i>mitten</i>
h	voiceless glottal fricative	h in the English name <i>Henry</i>
ʁ	voiced glottal fricative	h as in English between voiced sounds, as in the word <i>manhood</i>
y	high front rounded vowel	u in the French word <i>lune</i> (moon)
ɤ	high central unrounded vowel	as in the Russian word <i>сын</i> (son)
ɥ	High central rounded vowel	u as in the Norwegian <i>hus</i>
ɔ	high back unrounded vowel	u as in the Japanese name <i>Kazu</i>
u	high back rounded vowel	ou as in the French word <i>tout</i>
ʊ	upper mid-front rounded	ö as in the German name <i>Schönfeld</i>
ʊ	upper mid-back unrounded vowel	as in the Shan (Burma) word <i>ko</i> (salt)
o	upper mid-back rounded	o as in the English name <i>Mona</i>
ɪ	semi-high front unrounded vowel	y as in the English name <i>Lynn</i>
ɛ	lower mid-front unrounded	e as in the English name <i>Deborah</i>
ɐ	lower-mid front rounded vowel	œu as in the French word <i>œuf</i> (egg)
ɒ	lower-mid back unrounded vowel	u as in the English name <i>Tupperman</i>
ə	lower-mid back unrounded	o as in the English name <i>ford</i>
ɐ	open front unrounded vowel	a as in the English name <i>Hal</i>
ɐ	open central unrounded vowel	a as in the Portuguese word <i>para</i> (for)
ɐ	low front unrounded vowel	a as in the French word <i>pate</i> (paw)
ɑ	low central unrounded vowel	â as in the French name <i>Delâtre</i> or the word <i>pâte</i> (paste or dough)
ɔ	low back rounded vowel	o as in the British English word <i>hot</i>
ɐ	mid central unrounded vowel	e & a as in the English name <i>Belinda</i>
ʊ	semi-high back rounded vowel	u as in the English name <i>Butch</i>
ɛ	upper-mid front unrounded	a as in the English name <i>Mable</i>
ɪ	high front unrounded vowel	first e in the English name <i>Pete</i>
ɐ	rhotacized mid-vowel	ea as in the English name <i>Heather</i>

Appendix A: Descriptions of IPA Symbols (Continued)

Phonetic symbol	Description	Example
tʃ	voiceless alveo-palatal affricate	j as in the Chinese name Jin
tʃʰ	voiceless aspirated alveo-palatal affricate	q as in the Chinese name Qiu
ts	voiceless unaspirated dental affricate	ts as in the Chinese name Tsang
tsʰ	voiceless aspirated dental affricate	c as in the Chinese name Cao
ʙ	bilabial click	as in Southern Bushman languages
ɗ	dental (alveolar) click	as in Bushman
ʘ	palatal click	as in Bushman
ɘ	palato-alveolar click	as in Hottentot
ɬ	alveolar lateral click	as in Bushman, Zulu

Appendix B: Description of Phonetic Features

A. Major class features:

1. *Syllabic*

Forms the central peak of a syllable. Vowels are usually +syllabic, consonants are usually -syllabic, but some (like [l]) may be syllabic (as in "riddle")

2. *Sonorant*

Minimal constriction in the mouth. Vowels, as well as [n], [m], [r], [l], [w] are all +sonorant. Most other consonants are -sonorant.

3. *Consonantal*

Obstruction along a central point in the mouth. All English sounds except vowels and glides ([w] and [y]) are +consonantal.

B. Manner of Articulation Features:

4. *Continuant*

Continued air movement through the mouth during sound production. This feature contrasts fricative sounds like [f] and [v] with non-continuant sounds like [p] and [b].

5. *Strident*

Narrow obstruction through which air escapes, producing hissing or "white noise". [s], [z], [f], [v] and the sounds in church and judge are +strident. This is the most acoustically-based feature in this list.

6. *Delayed Release*

Gradual release of air. In English, it is used to distinguish the sounds in church and judge from [t] and [d].

7. *Nasal*

Soft palate at the back of the mouth is lowered and air goes into nose. In English, [n], [m] and [ŋ] (the final sound in king) are +nasal.

8. *Lateral*

Side(s) of tongue lowered so that air escapes along side, as in English [l].

C. Place of articulation:

9. *Anterior*

Obstruction of mouth anywhere from gum ridge forward to lips. English [p], [b], [m], [f], [v], and [θ] (as in the) are all +anterior.

10. *Coronal*

Front of the tongue raised. The sounds [t] and [d] are +coronal. Sounds like [k] and [g] are -coronal.

11. *High*

Body of tongue raised. [j] (as in yellow), and the vowel [ɪ] (as in feet) are -high.

Appendix B: Description of Phonetic Features (Continued)

12. Low

Body of tongue lowered. The vowels [ɒ] as in back and [ɜ] as in father are +low.

13. Back

Body of tongue moved back. The sounds [k] and [g] and the vowel [u] as in boot are +back.

14. Tense

Root of tongue muscle tensed. The vowel [ɛ] (as in feet) is +tense. The vowel [ɪ] as in fit is -tense.

15. Round

Lips pursed or rounded. English vowel [u] (as in boot) is +round, while [ɪ] (as in beet) is -round.

Appendix C: Phonetic Features for [p], [b] and [f]

Phonetic Features	[p]	[b]	[f]
syllabic	-	-	-
sonorant	-	-	-
consonantal	+	+	+
anterior	+	+	+
coronal	-	-	-
high	-	-	-
low	-	-	-
back	-	-	-
continuant	-	-	+
strident	-	-	+
delayed release	-	-	-
voiced	-	+	-
nasal	-	-	-
lateral	-	-	-
round	-	-	-

References

- Chomsky, Noam and Halle, Morris. *The Sound Pattern of English*, Harper & Row, New York, 1968.
- Cummings, D. W. *American English Spelling*, The Johns Hopkins University Press, London, 1988.
- Hanks, Patrick, and Hodges, Flavia. *A Dictionary of First Names*, Oxford University Press, Oxford, 1990.
- Hanks, Patrick, and Hodges, Flavia. *A Dictionary of Surnames*, Oxford University Press, Oxford, 1989-90.
- Hermansen, John C. *Automatic Name Searching in Large Data Bases of International Names*, unpublished dissertation, Georgetown University, Washington, D.C., 1985.
- Ladefoged, Peter. *A Course in Phonetics*, Harcourt Brace Jovanovich, Publishers, San Diego, 1982.
- Laver, John. *Principles of Phonetics*, Cambridge University Press, 1994.
- MacKay, Ian. *Phonetics: The Science of Speech Production*, Little, Brown, and Company, Boston, 1987.
- Symonds, Martin A. *Mandarin Pronunciation*, Taipei Language Institute, Taipei, 1986.
- Wolfram, Walt and Johnson, Robert. *Phonological Analysis: Focus on American English*, The Center for Applied Linguistics & Harcourt Brace Jovanovich, Inc., 1982.

Technology Demonstration System Narrative Description

February 26, 1998

Contract No. 97-F131000-000

Delivered to the Office of Research and Development



© 1998, Language Analysis Systems, Inc.
All rights reserved.

2214 Rock Hill Road—Herndon, VA—20170.

1.0 Introduction

This narrative describes the algorithms and techniques used by the Name Search - Technology Demonstration System (NS-TDS). It is the English-language version of the C++ source code that was used to develop the system. There are three major sections, covering NS-TDS support files, building the data base and performing a query. Each section relies on the contents of the previous section, so to effectively understand the system, this document should be read from beginning to end.

This narrative is tied to the source code through the use of paragraph numbers and comment lines. That is, whenever a block of code implements a technique or algorithm described in this document, a comment line has been inserted referencing the paragraph number. Comment lines are in the format, "// narrative paragraph number, x.x", where "x.x" stands for the paragraph number. If a block of code refers to more than one narrative paragraph, additional comments are added as separate lines.

2.0 Support Files

NS-TDS is a data-driven application dependent on a number of files that encapsulate years of computational linguistic research. These files represent the heart of the system and are essential to understanding how the primary algorithms work. This section of the narrative introduces these files by describing their purpose, contents and use.

2.1 Name Classifier Tables

TDS classifies the culture of a name as either Arabic, Chinese, Hispanic or "Other" (the default) by statistically analyzing its spelling. This analysis is accomplished with the aid of the following culture specific statistical distribution tables:

2.1.1 Digraph Score

Digraphs are contiguous letter pairs formed by parsing a name bracketed by a beginning and an ending boundary. For example, the name "FRED" consists of five digraphs: "#F", "FR", "RE", "ED" and "D#", where the symbol "#" represents a name boundary. In this table, digraphs that are clear indicators or contra-indicators of a particular culture are stored with a relative score. NS-TDS uses culture-specific tables to show the statistical likelihood of a particular digraph occurring in the applicable culture. For example, the digraph "QA" occurs almost exclusively in Arabic names, whereas the digraph "FM" almost never occurs in Arabic names. In the Arabic digraph table, "QA" is associated with a high positive score, and "FM" is associated with a low negative score.

2.1.2 Trigraph Score

Trigraphs are contiguous letter triplets that, for the purposes of TDS, are limited to the beginning and ending trigraphs. For example, the name "FRED" consists of the trigraphs "#FR" and "ED#". As with digraphs, NS-TDS uses culture specific tables to show the statistical likelihood of a particular trigraph occurring in the applicable culture.

2.1.3 Name Stop List

While generally good indicators of culture, digraph and trigraph distributions can erroneously classify specific names. For example, the name "BARKER" is identified as Arabic because it contains common Arabic letter patterns. The Name Stop List tables were implemented as a stopgap fix to this problem. For each culture, there is a Name Stop List table that contains a name along with a score that is either very positive (set to 2000) or zero (0). A high score means that name belongs to that culture; a score of zero means that the name does not belong there. So, "BARKER" is in the Arab Name Stop List with a score of 0.

The information in these tables is repeated for each culture and name part (i.e., given name or surname). For example, the following tables exist for Arabic:

agdi.dbf	Arabic digraph scores
agtri.dbf	Arabic trigraph scores
asdi.dbf	Arabic digraph scores
astri.dbf	Arabic trigraph scores

agnames.dbf	Arabic given name stop list
asnames.dbf	Arabic surname stop list

There are similarly named tables for Chinese starting with the letter "c" and tables for Hispanic starting with the letter "h".

2.1.4 Phonetic Rules

In order to convert the spelling of a name into a phonetic representation, NS-TDS consults several rule files. They contain records that consist of search parameters based on spelling and replacement regular expressions based on International Phonetic Alphabet (IPA) characters. Take the following rule, for example:

Boundary, "KN", Vowel, "(kn|kan|n)

It says that if the letter string "KN" is found at the beginning of a name ("Boundary") and is followed by a vowel, replace it with the IPA string, "(kn|kan|n)", where "|" indicates "or". The replacement string indicates that there are three possible pronunciations: [kn] or [kan] or [n]. (The use of square brackets is standard phonetic notation to indicate sounds rather than spelling). The spelling of a name is run through the rules until all characters are replaced with regular expressions. The name "KNOX" thus results in the regular expression (kn|kan|n)(a)(ks).

NS-TDS uses eight rule sets. For each of the four cultures (Anglo, Arabic, Chinese and Hispanic), there is a single vowel rule set and a multiple vowel rule set. The one vowel versions level all vowels to an [a] and produce fewer variations. They are used for retrieval. The multiple vowel versions contain three basic vowel sounds, [a], [i] and [u], and are used in the ranking of retrieved names, since they are more precise than single-vowel rankings.

2.1.5 Simplified Phonetic Rules

One additional phonetic rule file is maintained to aid in the filtering process. It is a cross-reference file between all of the possible replacement strings in the single vowel rule sets and a simplified version of the replacement string. It is "simplified" in the sense that all *unbalanced* "ors" become *balanced*. For example, the replacement string (kn|kan|n) is "unbalanced" in that the possible pronunciations can contain one, two or three sounds. The simplified version is, (k?)(a?)(n), where "?" means that the sound is optional. The simplified string allows TDS to compare two regular expressions that may generate thousands of possible pronunciations with one calculation, thereby improving performance dramatically. Note that the simplified strings sometimes generate more possible pronunciations than the original replacement string, but never fewer. The additional pronunciations are handled adequately by the Ranker (see Section 4.5.1). Currently, this file is named *ids.simp_rul*.

2.1.6 Leveled IPA Matrix

Generating retrieval keys requires the creation of leveled IPA variant strings. That is, similar sounds (i.e., [s] and [z]) are treated as a single set. NS-TDS uses a cross reference file to define the set relationships. It is currently called, *grouparray.dat*.

2.1.7 Feature Difference Matrix

One of the key components of TDS is the ability to calculate a phonetic score when comparing two names. When comparing individual sounds, the calculation weights the difference between two sounds based on a *feature distance matrix*. This matrix consists of all combinations of two IPA characters and a score between 0.0 and 1.0 representing their phonetic proximity to one another, as defined by articulatory measures of similarity. It also contains records that represent the insertion or deletion of an extra sound. For example, the score assigned to the replacement of a [t] with a [d] is lower than the score assigned to the replacement of a [t] with a [k]. Further, inserting a vowel is given a lower score than inserting consonant such as [t] or [k].

The scores contained in this matrix reflect penalties. That is, higher scores mean that the sounds are further apart. All of the scores are based on linguistic principles of articulation, and reflect the number and type of phonetic features that cause the sounds to be different.

3.0 Building the NS-TDS Data Base

In order to search a large number of names quickly, NS-TDS uses a data base of name information and indices. This data base is built by a program hereafter referred to as the *Data Loader*. This program takes as input a text file of names that are preceded with a group ID. The group ID is a minor component of TDS that was implemented to facilitate an independent evaluation by ORD.

Building the data base consists of two major steps. First, the names are pre-processed to generate the information needed by the retrieval, filtering and ranking algorithms. This pre-processed data is stored in temporary tables that are subsequently turned into the NS-TDS data base and indices. The following paragraphs describe this process in detail. Where appropriate, examples are used to make the description easier to understand.

3.1 Pre-Process Names

All names are pre-processed to ensure validity (see 3.1.1) and to gather the information necessary for retrieval, filtering and ranking. This process shares many of the components used to pre-process a query name during an NS-TDS search.

3.1.1 Edit the Name

Input names are provided to the Data Loader in a text file and are edited according to the following specifications: Positions 1 through 6 must contain a group ID, where the first character must be a digit or the letter "Z". All other characters must contain a digit. Position 7 must be blank. Positions 8 through 37 contain the name and can only consist of upper case letters or an apostrophe. Furthermore, the name must be at least 2 characters in length and no longer than 30 characters. Any records that fail to follow the prescribed format are rejected and written to an error log, along with an appropriate message.

3.1.2 Classify the Name

The spelling of the name is statistically analyzed to determine the probable culture (Arabic, Chinese, Hispanic, or "Other"). This analysis is accomplished with the aid of the name classifier tables.

First, the name is parsed into digraphs (contiguous letter pairs) and beginning and ending trigraphs (contiguous three letter triplets) (see 2.1.1 and 2.1.2). Next, the digraphs and trigraphs are located in the appropriate classifier table to obtain the individual score. All of the scores are summed to obtain a total score. This process is repeated for all cultures.

Then the Name Stop List tables for each culture are checked. If the name is found in one of the tables, the associated score is returned ("2000" means in the culture, "0" means not in the culture). If the name is found, the previously calculated culture score is replaced.

Finally, each score is compared to a culture-specific threshold. If no scores exceed the culture threshold, the name is classified as "Other". If one score exceeds the appropriate threshold, the name is classified accordingly. If more than one score exceeds the culture threshold, the highest score is chosen and that culture is returned. If there is a tie (very unlikely), the culture is chosen

alphabetically with Arabic first followed by Chinese and then Hispanic. It is important to note that an input name will receive only one classification.

3.1.3 Generate 1 Vowel Regular Expressions

In this step, the spelling of the name is run through the spelling-to-IPA phonetic conversion rules, to generate a regular expression that represents all of the possible pronunciations of the name. Every name is run through the single-vowel Anglo phonetic rule set, which is the default/generic rule set. If the name was classified as Arabic, Chinese or Hispanic, it is also run through the appropriate single-vowel rule set for that culture, generating a second IPA regular expression.

3.1.4 Generate Simplified Regular Expressions

Using the simplified phonetic rules, an simplified regular expression is generated. The expression is encoded into compact byte representations to make further calculations faster. As before, if the name was classified as Arabic, Chinese or Hispanic, a second simplified regular expression is generated according to the appropriate rule set.

3.1.5 Generate 1 Vowel Variants

Using the generated regular expression, a list of possible IPA variants is generated and added to a temporary table of variants for all input names. As an example, the name "KNOX", which generates the regular expression (kn|kan|n)(a)(ks), generates the following variants: [knaks], [kanaks], and [naks]. The temporary table lists all variants, as well as the name that generated the variant. It is used later in the data base build process.

3.1.6 Determine the Initial Consonants

The variants are then analyzed, to generate a list of all possible name-initial IPA consonants. For example, the name "KNOX" starts with the regular expression (kn|kan|n), which can have an initial consonant of [k] or [n]. Note that if the variant starts with an IPA vowel, the first IPA consonant is used to build this list. Thus the name O'NEIL would have [n] as the initial consonant. This information is used by the Ranker (see section 4.4.4 below).

3.1.7 Set the Initial Vowel Switch

Next, the variants are analyzed to determine if it is possible for the pronunciation of the name to start with a vowel. It is a three-way switch that indicates whether the pronunciation (1) can never start with a vowel, (2) can sometimes start with a vowel or (3) always starts with a vowel. This information is used by the Ranker (see section 4.4.5 below).

3.2 Build Data Base and Indices

This step takes all of the information produced during pre-processing and builds the data base and indices used by TDS for retrieval, filtering and ranking.

3.2.1 Create Name Files

A name file is generated for all four cultures processed by TDS (Anglo, Arabic, Chinese and Hispanic). "Anglo" represents the default, and therefore all names generate an Anglo record; only those names that are appropriately classified generate records in the other culture name files. Each record in the name file contains: the spelling of the name, the simplified regular expression codes, the list of initial consonants, the initial vowel switch, the group ID and an internal unique ID.

The naming convention is a four-letter culture identification followed by the extension, "nam". Currently, the following name files are generated: *angl.nam*, *arab.nam*, *chin.nam* and *hisp.nam*.

3.2.2 Generate Leveled Variants

Using the variants generated during pre-processing and the leveled IPA matrix, a list of leveled variants is built. Furthermore, the input variants have duplicate contiguous characters removed. The IPA characters in "KNOX" generate the following numeric codes, based on sets of similar sounds: [k] = 5; [n] = 2; [a] = 0; [s] = 4; [z] = 4. (Note that [s] and [z] are both indexed as "4", since they are similar sounds). The following unique leveled variants are generated: 52054, 502054 and 2054. Note that the number of leveled variants is usually less than the number of non-leveled variants. For each input name, the leveled variants are added to a temporary file that lists all leveled variants and the name that generated it.

3.2.3 Create Retrieval Indices

Retrieval indices consist of a unique sorted list of leveled variants. As with the name files, one index is generated for each culture. Each index is created by sorting and then *deduping*, i.e., removing duplicate forms from the previously-built temporary file of leveled variants.

The files produced by this step are named *angl.idx*, *arab.idx*, *chin.idx* and *hisp.idx*.

3.2.4 Create Index-to-Name Maps

Finally, a map file is created that cross-references all of the index records with the name records that generated the leveled variant. The retrieval index records contain a pointer to a map record. The map record contains a list of pointers to name records. This structure allows TDS to quickly scan the indices and generate a list of candidate names during retrieval.

The names of the map files are *angl.vec*, *arab.vec*, *chin.vec*, and *hisp.vec*.

4.0 Performing a Query

The heart of NS-TDS is the ability to perform a query that returns a ranked list of results. This is done using five major steps: query pre-processing, exact phonetic search, similar phonetic search, initial ranking and final ranking. The following paragraphs describe these steps and their components in some detail. When appropriate, examples are used to make the descriptions easier to understand.

4.1 Pre-Process Names

The query name is pre-processed to ensure validity and to gather the information necessary for retrieval, filtering and ranking. This process shares many of the components used to pre-process an input name during the building of the NS-TDS data base.

4.1.1 Edit Name

After the user enters a query name via the user interface, it is edited according to the following criteria: The name can only consist of the 26 letters of the Roman alphabet or an apostrophe. Further, the name must be at least 2 characters in length and no longer than 30 characters. Errors are displayed to the user in a dialog box, along with an appropriate message.

4.1.2 Classify Name

If the user has specified that culture classification is automatic (the default), the spelling of the name is statistically analyzed to determine the probable culture (Arabic, Chinese, Hispanic, or "Other"). This analysis is accomplished with the aid of the previously described name classifier tables (see 2.1 above). If the user has overridden the default and manually specified a culture, this step is skipped.

The name is then parsed into digraphs (contiguous letter pairs) and beginning and ending trigraphs (contiguous three letter triplets). Then, the digraphs and trigraphs are located in the appropriate classifier table to obtain the individual score. All of the scores are summed to obtain a total score. This process is repeated for all cultures.

Next, the Name Stop List tables (see 2.1.3 above) for each culture are checked. If the name is found in one of the tables the associated score is returned ("2000" means in the culture, "0" means not in the culture). If the name is found, the previously calculated culture score is replaced.

Finally, each score is compared to a culture specific threshold. If no scores exceed the culture threshold, the name is classified as "Other". If one score exceeds the appropriate threshold, the name is classified accordingly. If more than one score exceeds the culture threshold, the highest score is chosen and that culture is returned. If there is a tie (very unlikely), the culture is chosen alphabetically, with Arabic first, followed by Chinese and then Hispanic.

It is important to note that an input name will receive only one classification. Further, if the classification is Arabic, Chinese or Hispanic, all further pre-processing will be performed twice (once for the default Anglo culture and once for the culture identified by classification or as manually specified by the user).

4.1.3 Generate 3 Vowel Regular Expressions

In this step, the spelling of the name is run through the multiple vowel phonetic rules to generate a state table that represents all of the possible pronunciations of the name in IPA form. Every name is run through the default Anglo phonetic rule set. If the name was classified as Arabic, Chinese or Hispanic, it is also run through the appropriate rule set for that culture generating a second state table. These will be used during the *exact* phonetic search (see 4.2).

4.1.4 Generate Multiple (3) Vowel Variants

Using the multiple vowel state table, a list of all possible IPA variants is generated. As an example, the name, "KNOX", which generates the regular expression (kn|kan|n)(a|u)(ks), generates the following IPA variants: [naks], [nuks], [kanaks], [kanuks], [knaks], and [knuks]. This list will be used to perform a brute force phonetic score adjustment on names that pass preliminary ranking (see 4.5).

4.1.5 Generate 1 Vowel Variants

Using the 1-vowel state table, a list of all possible one-vowel IPA variants is generated. As an example, the name, "KNOX", which generates the regular expression, (kn|kan|n)(a)(ks), generates the following variants: [naks], [kanaks], and [knaks]. This list will be used to generate retrieval and ranking information.

4.1.6 Determine the Initial Consonants

The single-vowel variants are then analyzed to generate a list of all possible initial IPA consonants. For example, the name KNOX starts with the regular expression, (kn|kan|n), which can have an initial consonant of [k] or [n]. Note that if the name starts with a vowel, the first IPA consonant is used to build this list. Thus, the name, O'NEIL would have [n] as the initial consonant. This information is used during ranking (see 4.4.4).

4.1.7 Set the Initial Vowel Switch

Next, the single-vowel variants are analyzed to determine if it is possible for the pronunciation of the name to start with a vowel. It is a three-way switch that indicates that the pronunciation (1) can never start with a vowel, (2) can sometimes start with a vowel, or (3) always starts with a vowel. This information is used during ranking (see 4.4.5).

4.1.8 Generate Leveled Variants

Using the appropriate one-vowel rule set, a temporary list of all possible IPA variants is generated. This list, along with the leveled IPA matrix, is used to build a list of leveled variants. Also note that duplicate contiguous characters are removed. For example, the IPA characters in KNOX generate the following numeric codes, based on sets of similar sounds: [k] = 5; [n] = 2; [a] = 0; [s] = 4; [z] = 4. (Note that [s] and [z] are both indexed as "4", since they are similar sounds). The following unique leveled variants are generated: 52054, 502054 and 2054. Note that the number of leveled variants is usually less than the number of non-leveled variants.

4.1.9 Generate Simplified Regular Expressions

Using the simplified phonetic rules, a simplified regular expression is generated. The expression is encoded into compact byte representations to make further calculations faster. As before, if the name was classified as Arabic, Chinese or Hispanic, a second simplified regular expression is also generated.

4.1.10 Initialize Search Parameters

The search parameters specified by the user or defaulted by the application are stored along with the query information. These parameters set thresholds for retrieval and filtering, and determine the weights given to individual ranking scores.

4.2 Exact Phonetic Match

An exact phonetic search is always performed by TDS. It is a quick search that retrieves names which share at least one possible pronunciation with the query name and passes them to the ranker. A search of the Anglo data base is always performed; if a non-Anglo culture was determined or specified by the user, the search is repeated for the appropriate culture.

4.2.1 Retrieve Candidates

Each of the leveled variants generated by pre-processing the query name are used as a key to perform a binary search of the retrieval index, which is a set of unique leveled variants for the name data base. In the case of "KNOX", three leveled variant indices are retrieved: 2054, 502054 and 52054.

4.2.2 Retrieve Name Information

Using the index-to-name map files, all data base names and associated information that could possibly generate the leveled variants found above are put into a list. In the case of "KNOX", names such as "NOCKS", "NOX", "KNOCKS" and "NAUCHS" are returned.

4.2.3 Execute Exact Phonetic Match Algorithm

For each name retrieved, a regular expression is generated using the appropriate multiple-vowel rule set. Each of these is compared to the query's regular expression to determine if there is an intersection. In other words, the two expressions are evaluated to see if they can generate a matching variant. This evaluation is done by generating non-deterministic finite state tables and walking through each table until a match is impossible (i.e., the names do not match), or the end of both tables is reached (i.e., the names match). If the name passes this algorithm, it is placed in a list.

4.2.4 Pass Exact Matches to the Ranker

All names that pass the exact-match algorithm are sent to the Ranker, along with the information retrieved from the name file. In addition, the phonetic score is set to 1.0, which is the highest

possible score, and the pipe (rule set) that was used to retrieve the name is passed. Note that if a name was found to be an exact match under two cultures, it is included twice.

4.3 Similar Phonetic Match

A similar phonetic search is performed only if the user has requested it. Note that the default is to perform a similar search. It is slower and more thorough than the exact search, and retrieves names that sound similar (based on principles of articulation) to the query name to the ranker. A search of the Anglo data base is always performed; if a non-Anglo culture was determined or specified by the user, the search is repeated for the appropriate culture.

4.3.1 Scan the Retrieval Index

A complete scan of the retrieval index is performed, and each leveled variant is compared to the leveled variants generated by pre-processing the query. The comparison uses a standard edit distance calculation to determine how far apart two strings are. The algorithm determines the minimum number of edits (insertion, deletion or replacement of IPA characters) necessary to convert one string into another. A score is calculated by dividing the number of edits by the maximum length of the two strings and subtracting this fraction from 1 resulting in a score between 0.0 and 1.0. This score is compared to the retriever threshold, and those records with a score greater than or equal to the threshold are added to a candidate list.

4.3.2 Filter the Candidates List

This list is scanned and, if the name has not already been retrieved by the exact match algorithm, the simplified regular expression of the query name is compared to simplified regular expressions of all of the candidate names. This comparison uses a more linguistically sophisticated edit distance algorithm that takes into account the phonetic features of each sound. All edits are weighted according to the relationships stored in the Feature Distance Matrix. For example, the replacement of similar sounds that share most phonetic characteristics, like [s] and [z], are given a small penalty. The "cost" of replacements is determined by where and how sounds are articulated in the mouth and the "effort" required to produce one rather than the other. Similarly, some insertions and deletions of sounds are more costly than others (e.g., insertion of a [t] is more costly than insertion of a vowel, [a]). So, instead of computing the minimum number of edits required to convert one string into another, this algorithm calculates the path of least resistance. As with the retrieval calculation, a score between 0.0 and 1.0 is obtained by dividing the total penalty by the maximum length and subtracting this fraction from 1. This score is compared to the filter threshold, and those records whose score is less than the threshold are discarded.

Finally, it is important to note that because simplified regular expressions can generate more variants than the expressions they were derived from, it is possible for this score to be higher than expected, although it is impossible to obtain a lower score. This deficiency is corrected during ranking.

4.3.3 Pass Similar Matches to the Ranker

For all records that pass the filter algorithm, additional information is gathered from the name file via the index-to-name map. Also, the phonetic score of these names is set to the score

calculated during the filter edit distance calculation, and the culture pipe (i.e., rule set) that was used to retrieve the name is passed. This list is sent to the Ranker for initial scoring. Note that if a name was found to be a similar match under two cultures, it is included twice.

4.4 Initial Ranking

Names that pass the retrieval and filter stages via the exact or similar phonetic match searches are sent to the Ranker, along with the phonetic score calculated by the filter and all of the data built during the pre-processing stages (initial consonant, initial vowel, etc.). The ranker also knows which search (exact or similar) produced the return. It takes this information, calculates several other scores, applies weights to those scores based on the query parameters and produces a ranked list of names with combined scores. Initial ranking differs from final ranking in that it uses the phonetic score calculated by the filter. Final ranking, which is described below (see 4.5), performs a more exhaustive and exact phonetic score calculation.

4.4.1 Calculate the Spell 1 Score

Because spelling is a relevant factor in determining similarity of names, the Ranker is set up to consider spelling in its calculations in ranking of names passed to it. In the case of exact matches, for example, all phonetic scores are 1.0, but spellings can vary widely (e.g., "LI", "LEE", "LEIGH"). The *Spell 1* score is a comparison of all the letters in the query name to all of the letters in the data base name. Each letter that matches contributes to the score, and no letter can be used more than once. Note that the position of the letter has no bearing on the score. So, the "K" in "KNOX" matches the "K" in "SACK". In addition, there is an option to bias the score so that letters on the left side of the query name count more than those towards the end. The "left-bias" factor defaults to true. A score is calculated by dividing the value of the matches (an integer, if left bias is not used) by the maximum length of the query and data base name and then subtracting this fraction from 1.0. This results in a score between 0.0 and 1.0.

4.4.2 Calculate the Spell 2 Score

The *Spell 2* score works similarly to *Spell 1*, except that it uses *digraphs* instead of single letters. Digraphs are contiguous letter pairs formed by parsing a name bracketed by a beginning and an ending boundary. For example, the name "FRED" consists of five Roman character digraphs: "#F", "FR", "RE", "ED" and "D#", where "#" represents a name boundary. Digraphs build some contextual information into the calculation, with the result that "FRED" and "BRID", which share two non-contiguous letters, have a lower *Spell 2* score than a *Spell 1* score. *Spell 2* uses the same left bias parameter and the same method to turn the calculation into a decimal number between 0.0 and 1.0 as *Spell 1*. Finally, the *Spell 2* score contains a special adjustment for names that start with a vowel. For example, when comparing the name, "NEIL" to "ONEIL" and "SNEIL", the score for "NEIL" will be adjusted upwards by a small factor.

4.4.3 Calculate the Syllable Score

The Syllable score compares the number of syllables in the query name to the number of syllables in the data base name. Counting the number of syllables in a name is based on the spelling, and essentially says that a syllable occurs when there are one or more vowels in a row, preceded by a consonant or a word boundary. Adjustments are made for special cases such as, diphthongs (multiple consecutive vowels pronounced as two syllables) and "E" or "ES" at the end

of name which often does not produce a separate syllable. A score is produced by dividing the difference in the number of syllables between the query name and the data base name by the maximum number of syllables in the query or data base name and subtracting the resulting fraction from 1.0. This results in a score between 0.0 and 1.0.

4.4.4 Calculate the Initial Consonant Score

The initial consonant sound in names hold particularly prominent positions in determining similarity of names. The Initial Consonant score compares the first occurrence of an IPA consonant in the query name to that of the data base name. The consonants are compared based on the Feature Distance Matrix, producing a score between 0.0 and 1.0. For example, [s] and [z] will return a high score, whereas, [k] and [r] will return a low score. Note that the first consonant can be different as is the case with the name, "KNOPF", which could start with a [k] sound or an [n] sound. The algorithm compares all possibilities and returns the best possible score.

4.4.5 Calculate the Initial Vowel Score

The Initial Vowel score comes into play when both the query name and the data base name start with a vowel. If this is not the case, the initial vowel score is 1.0. Otherwise, the IPA vowel or vowels that start the names are compared and a score is returned based on the feature distances between them. The score is a decimal between 0.0 and 1.0.

4.4.6 Calculate the Culture Score

The culture score compares the culture of the query name as determined by the classifier or as specified by the user with the "pipe"(rule set) used to retrieve the data base name. So, if a name is classified as Arabic, and the name being ranked was passed to the ranker via the Arabic pipe, the culture score is 1.0. If the query culture does not match the pipe used to retrieve the name, the culture score is 0.0. This allows the Ranker to "bump up" names that share the same cultural identity, as in Chinese "CHIN" and "CHANG" (versus non-Chinese "CHAIN", for example).

4.4.7 Calculate the Final Score

The final or total score is an amalgamation of all of all the previous scores. NS-TDS maintains a set of parameters that allows the user to assign weights to each of the various individual scores. Note that in the user version, these weights are not modifiable. The weights are intended to be percentages, so that each factor is a decimal between 0.0 and 1.0 and the total adds up to 1.0. This is not a requirement, as the calculation recomputes the weights relative to one another.

So, to arrive at the final score, all of the individual scores are multiplied by their weight and the results are summed. This results in a decimal score between 0.0 and 1.0 with a higher score indicative of a better match.

4.4.8 Set the Ranking Order

The absolute ranking of data base names is based on whether or not the name is an exact phonetic match and on the value of the final score. Exact phonetic matches are always ranked first, followed by similar matches. Within these two categories, names are ranked according to

the final score, with higher scores ranked at the top of the list. It is quite possible that an exact match will receive a lower final score than a similar match (if its spelling and/or culture scores are low, for example), and yet be ranked above the similar match based on its category of "exact match". For example, "KNOX" returns the exact match "NAX" with a lower score (.825) than the similar match "KNAGGS" (.848), but forces *all* exact matches, including "NAX", to the top of the list.

4.4.9 Return a Ranked Set

Finally, the Ranker eliminates names that do not meet or exceed the threshold set in the NS-TDS parameters, unless the name is considered an exact match. Exact phonetic matches are always returned, regardless of their score.

4.5 Final Ranking

The purpose of Final Ranking is to incorporate a more accurate phonetic score into the overall ranking. Recall that the phonetic score used by the initial Ranker is calculated by the Filter, using simple regular expressions. This algorithm, while fast, can inflate the phonetic score, producing inaccurate ranking. Further, the Filter calculates using single vowel rules, which can introduce another source of inaccuracy (e.g., "LITZ" = "LUTZ"). Final ranking adjusts the phonetic score by performing a brute force edit distance calculation, using multiple vowel rules. This calculation is performed at this point because it is time-consuming, and must be limited to the smallest possible set of input data to meet performance requirements. Note that Final Ranking is a parameter option, although the default is set to true.

4.5.1 Recalculate the Phonetic Score

All names that were retrieved via the similar phonetic search and passed initial ranking are reprocessed to produce an accurate phonetic score. Names that were retrieved via the exact phonetic match search do not need to be recalculated because their phonetic score is always 1.0. First, the names are passed through the appropriate cultural multiple vowel rule set to produce a list of all possible IPA variants. Then, a brute force edit calculation is performed; every variant from the query name is compared to every variant of the data base name by performing a phonetic edit distance calculation. The best score is retained and assigned to the result.

4.5.2 Recalculate the Final Score

Using the same logic as that used by the initial Ranker, the Final Score is calculated using the new, more precise phonetic score. This will result in lower scores for some names; if these names fall below the final score threshold, they are removed from the ranked list.

4.5.3 Rebuild the Ranked Set

Finally, using the new score, the set of names is ranked again; with exact matches at the top followed by similar matches. Also, the final ranker will only return the maximum number of names requested by the user. The default setting is 145. Thus, it is possible for a name to pass NS-TDS, but not be displayed on output.

Technical Plan
Technology Demonstration System
September 10, 1997
(amended)

Contract No. 97-F131000-000

Delivered to the Office of Research and Development



Language Analysis Systems, Inc.
2214 Rock Hill Road—Herndon, VA—20170

1.0 Introduction

This Technical Plan describes LAS's proposed design for the Technology Demonstration System (TDS) and includes a conceptual design, the target hardware platform, operating system, support software and development environment, the name data base and a work plan that provides a schedule for development and implementation.

2.0 Background

The TDS project is the result of the findings and recommendations of the Name Search Research Project, conducted from September, 1995 through June, 1997. The goal of the Project was to determine the utility and feasibility of using phonological information about pronunciation of person names in order to improve the quality of non-exact automatic name searching. Phase 1 of the Project concluded that there was substantial evidence to support the use of phonological information in automatic name searching. Specifically, the conclusions recommended:

- using the International Phonetic Alphabet (IPA) to represent multiple pronunciations of names unambiguously, and
- measuring articulatory similarity of names through phonetic features and processes.

Phase 2 of the Project built upon the results of Phase 1, specifically by:

- expanding, refining and testing sets of IPA rules from Phase 1 to represent multiple pronunciations of Anglo, Arabic, Hispanic and Mandarin Chinese names; test results returned at a retrieval rate of 92%;
- exploring a set of factors that contribute to articulatory similarity, including factors at the syllable level.

Phase 2 recommended the development of a Name Search Technology Demonstration System (TDS) to extend and transfer the phonology-based technology from the Name Search Research Project to a functional, automatic, integrated TDS.

3.0 Environment

The hardware and software environment are well defined. It is a simple environment that is geared to flexibility and performance. In other words, LAS does not intend to introduce complications by using resource intensive and/or expensive support software or hardware. TDS will be built using the following hardware and software components:

- Standard, high performance Intel-based laptop computers. By purchase time, we expect to configure the machines as follows:
 - Intel Pentium, Pentium Pro or Pentium II CPU;
 - 160 to 512 Mb of memory (160 is the current maximum);
 - 3 Gb of disk storage;
 - High speed CD-ROM (8x minimum);
 - Standard Ethernet network card for high speed data transfer;
 - High resolution monitor with a bright clear screen.
- Windows 32-bit operating environment and development software:
 - Windows 95 or Windows NT 4.x (depending on the processor available);
 - Microsoft Visual C++ version 5.x (for development only);
 - Microsoft Access (support table maintenance);
 - Custom data storage techniques maximizing memory usage (no RDBMS);
 - Multi-threaded architecture to begin displaying responses within 12 seconds.

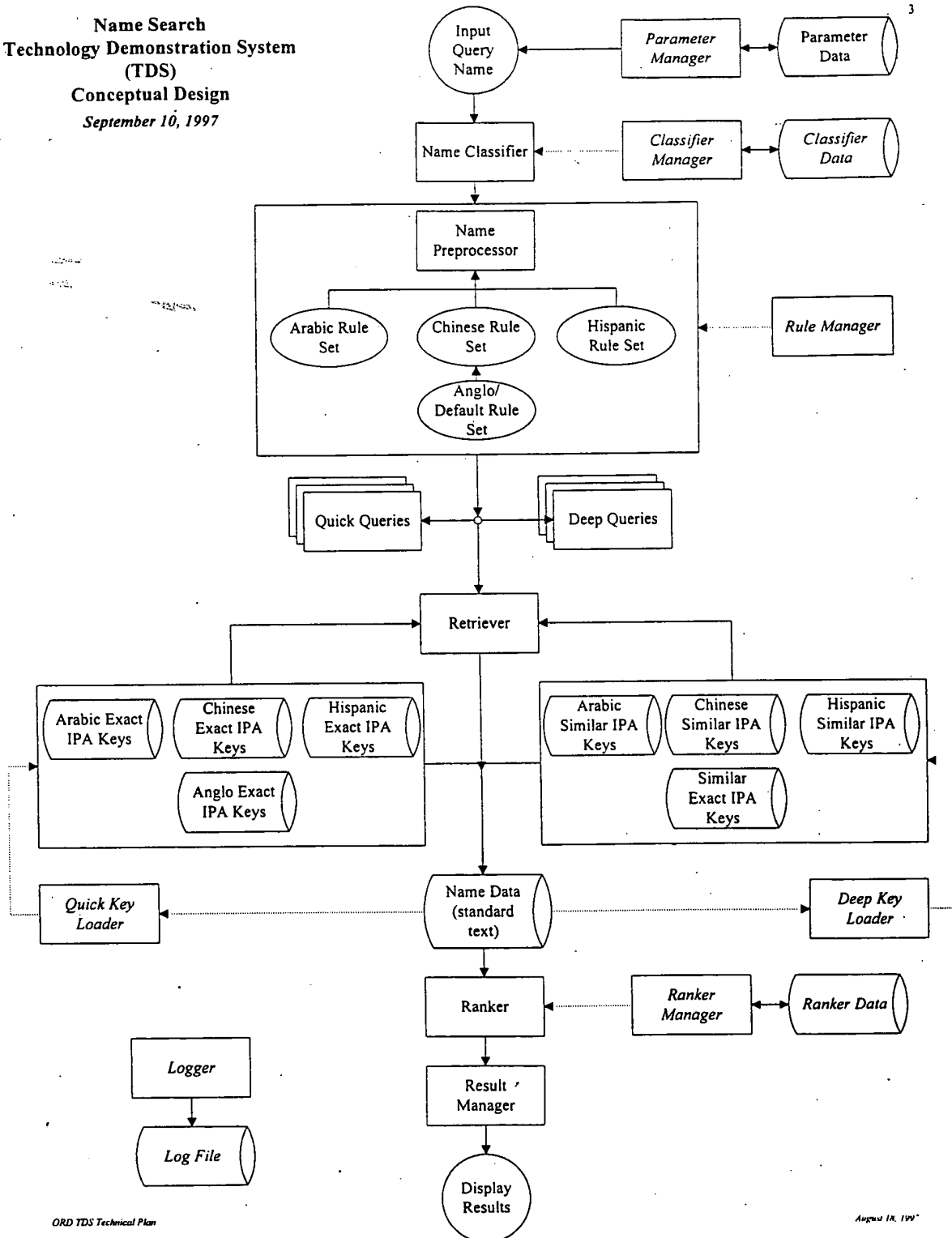
TDS will run on a top-of-the-line standard laptop computer, a suite of custom developed executables and dynamic link libraries (DLLs) and standard end user software (i.e., MS Access, Excel, etc.). There will be no special or extraordinary system or software requirements.

4.0 Conceptual Design

The following diagram gives an overview of the system LAS intends to build. It is based on previous documents developed by the sponsor and LAS and represents research done to date. It identifies the major components and processes to be included in TDS; however, it is a preliminary design, and is subject to change depending on the outcome of further research and time restraints. After the diagram, each of the components and interactions between them is described in further detail.

In the chart, components that are identified in *italics* are support functions not intended for standard use. They are, however, necessary for LAS to build, tune and test the system.

**Name Search
Technology Demonstration System
(TDS)
Conceptual Design
September 10, 1997**



Input Query Name - A simple Graphical User Interface (GUI) will be built to allow the user to enter a name that will be compared to the name data base.

Parameter Manager - LAS will provide options so the user can tune or limit the search. These options will be controlled through the GUI and stored in a **Parameter Data** set. Parameters currently being considered include:

- exact only (fast) or exact and similar matches;
- level of similarity (loose or tight);
- culture specific matches (Anglo, Arabic, Chinese and/or Hispanic);
- number of returns (maximum/default = 145);
- bypass name classification.

Note that it will not be necessary to set parameters to perform a name search. Default parameters will be used in the event that the user goes directly to the name check screen.

Name Classifier - The name classifier determines whether the ethnicity of a name is Arabic, Chinese or Hispanic. The LAS name classifier uses a data base of contiguous letter pairs (digraphs) and triplets (trigraphs) that has been statistically analyzed to rank digraphs and trigraphs according to ethnic origin. With this information, it calculates a score for each culture that shows the probability of the name being Arabic, Hispanic and/or Chinese. The highest positive score will determine which non-Anglo algorithm to use in addition to the standard Anglo algorithm. Note that it is possible for all scores to be negative, in which case only the Anglo algorithm will be used. This component will be based on an existing system developed in Clipper by LAS that will be converted to C++ to better interface with the other components.

Classifier Manager - This is a simple interface necessary to apply values to the digraphs and trigraphs according to ethnicity. Most likely, LAS will use a standard data base package to manipulate classifier data (i.e., MS Access). Note that the existing classifier data base is already returning adequate results. Improvements will be made if time and resources permit.

Name Preprocessor - At a minimum, this component will convert the input name into one or more IPA representations. Almost certainly, it will generate numerous variants based on different phonetic representations that will be passed to the **Retriever**. Furthermore, additional information about the query name will be necessary in order to use the similar search keys (i.e., name length, syllabic structure, etc.).

Rule Sets - Four rule sets will be used to convert Roman character representations of names into IPA representations. The default rule set, Anglo, will always be used; the other three, Arabic, Chinese and Hispanic will be used if the name is classified as belonging to one of these ethnic groups and the user has specified that other ethnic variations are to be used. These rule sets will be based on the work done in previous projects. The Anglo rule set will need considerable

modification to support Anglo pronunciations of non-Anglo names. They will be maintained by a **Rule Manager**, that allows LAS to build and modify rule sets as necessary.

Quick Queries - To ensure that the 12 second initial response time requirement is met, LAS intends to segment and multi-thread searches of the name data base. Quick queries retrieve those records that contain the same IPA characters or the same IPA consonants with a vowel place-holder, or the same IPA consonants. The ultimate retrieval scheme will be determined by further research. This approach will allow TDS to pass a small subset of data to the Ranker and begin returning most of the "best" names quickly. Note that this scheme does not consider differences in name length (i.e., insertion and deletion). The output of the quick query component will be a list of IPA representations that the **Retriever** will use to extract records for evaluation by the **Ranker**.

Deep Queries - By far the most difficult problem to solve, deep queries will allow TDS to subset the name data base into phonetically similar sections and account for varying levels of name and possibly syllable length. They must consider the insertion and/or deletion of IPA characters and the proximity of different IPA characters based on the number and importance of features they have in common (e.g., "p" and "b" differ by only one phonetic feature). Almost certainly, deep queries will include all names retrieved by quick queries. If performance is acceptable for deep queries, the quick query logic may become unnecessary. The output of the deep query component will be a list of IPA representations that the **Retriever** will use to extract records for evaluation by the **Ranker**.

Retriever - This component accepts query lists from the query preprocessors and passes subsets of the name data base to the **Ranker**. Operating simultaneously with the query components, it processes query lists in the order that they are received. Each input list will be identified as a "quick" or a "deep" list so that the component can choose the proper key set to use to generate the output list. Once the subset is determined, the retriever will build a list or a range of records to be passed to the **Ranker**. This list will contain the IPA representation used to retrieve the record, the actual Roman character representation of the name and the rule set used to return the name.

Quick Keys - Each name in the data base will generate one or more IPA representations of the Roman character version. Each rule set can generate different IPA representations. All representations will be stored in the **Quick Key** data set that will point to the name that generated the particular version. Furthermore, quick keys will be tagged as belonging to the rule set that generated the representation.

Deep Keys - This data set will consist of keys that contain IPA representations, IPA similarity, name length and possibly, syllabic information. It will be designed to allow for subsetting of the name data base into names that are potentially similar to the query name. It must overcome the two major problems in determining name similarity: sounds can be mispronounced (Pine = Bine) and names can be substrings of each other (McDonald = Donald). A key area of research that must be resolved early in the project is the use of indices to represent similar IPA characters (one character to represent "b" and "p").

Name Data - This component represents the raw data provided by LAS internally for development and ultimately by the sponsor for the production version of TDS. Each name will be stored in its Roman character representation and will be identified by a record ID. These ID's will be used to tie the raw name to the quick and deep keys.

Key Loaders - Batch programs will be developed that take an ASCII text file of names as input to generate the **Name**, **Quick Key** and **Deep Key** data bases. This program will use the **Name Classifier** and **Name Preprocessor** to generate keys and build or rebuild these data bases. It will edit the names and produce a summary statistical report and a detailed error report showing any abnormalities encountered (i.e., invalid length, invalid characters, etc.).

Ranker - This component processes a list of candidate records generated by the **Retriever**. The list will consist of records containing the IPA key that returned the record, the rule set used to generate the IPA key and the actual name. The Ranker will sort the names in order of match quality based on parameters set in the **Ranker Manager** data set. Output will be passed dynamically to the Result Manager for real-time display to the user. Ranking methods will be based on schemes developed in phase 2 of the phonology project (regular expression intersection and the "voter scheme"). It will also consider the rule set used to regularize the input name into IPA representations.

Ranker Manager - This is an optional component that will allow LAS and/or the sponsor to rank returns according to different sort schemes. As mentioned above, the ranking schemes will be based on previous work: regular expression intersection or a voter scheme, and possibly, non-phonetic schemes such as: Soundex, digraph analysis, edit distance methods, etc.

Result Manager - This component will accept input from the **Ranker**, and maintain a deduped, sorted list based on the parameters set by the **Ranker Manager**. This list will be passed to the GUI for display to the user, and it will be managed dynamically so that the list is constantly being updated as results are processed by the **Ranker**.

Display Results - This component is the output side of the GUI. It displays the list produced by the **Ranker** for viewing and other manipulation (printing, saving to a file, etc.) by the user. The outputs are maintained by query name and are updated dynamically as results are returned from the **Ranker**. In addition to the ranked list of names, the GUI will also display information on why a particular name was chosen and will be given a score that relates to names above and below it on the list.

Logger - This component will be a development and debugger tool for LAS to determine how well TDS is working, and to aid in testing and problem resolution.

5.0 Name Data Base

The ultimate target for TDS is a sponsor data base consisting of 3 million unique name segments (i.e., "John" and "Fitzgerald", not "John Fitzgerald"). LAS must generate a similar data base of name segments since the sponsor data base is classified. To do this, LAS will take advantage of numerous resources that will be used without compromising the privacy and sensitivity of the data. That is, only name segments will be extracted from these sources. It will be impossible to tie the TDS names to the source data base. Sources to be used include:

- Visa Lookout Data from the Department of State;
- Passport Lookout Data from the Department of State;
- Census Data from the Department of Commerce;
- Phone Book data from commercial sources;
- Known variant lists.

Should the above sources fail to generate 3 million unique name segments, LAS will resort to generating variations by programmatically manipulating letter variations (i.e., "ck for "ch", "e" for "i", etc.). Currently, LAS has processed 20 million names, which have generated 1 million unique name segments.

Crucial to the successful completion of TDS is an opportunity by LAS to evaluate sponsor data as soon as possible. While not in the Statement of Work or the Project Plan, LAS feels it is advantageous to the sponsor to allow LAS to gain access to the sponsor name data base as soon as possible. While no problems are expected, it is prudent to verify this assumption as the success of TDS is ultimately dependent on the ability to successfully integrate sponsor data.

6.0 Work Plan

Please note that this section of the Technical Plan has been copied in entirety from the Project Plan previously submitted. Attachment A to this plan is a Gantt chart with a Work Breakdown Structure that describes the schedule of development LAS intends to follow. The rest of this section describes the major events in the Gantt chart.

The schedule for the development of TDS spans eight months and consists of four major phases:

- **Planning** - One month to generate project and technical plans.
- **Phase 1 Development** - Three months to resolve research issues, determine a strategy to find "similar-to" names, define and validate linguistic search techniques, and produce a limited version of TDS for an early look test.
- **Phase 2 Development** - Three months to expand phase 1 into a fully functional system to include expanded rule sets that enable TDS to accommodate Anglo pronunciations of foreign names and native pronunciations for Arabic, Hispanic and Chinese names.

- **Implementation** - Four months (with three months overlapping the development effort) to procure two laptop computers, install the system for the sponsor, provide training, and document the results of the project.
- **Maintenance** - Four months to modify, upgrade and correct TDS at the direction of the sponsor.

Each phase concludes with a specific set of deliverables (both internal to LAS and formal deliveries to the sponsor). There is some flexibility in the schedule, however, the dates set for sponsor deliveries are firm.

6.1 Phase 1 Development

The purpose of phase 1 is prove that LAS can develop a viable name search system based on phonetics. The goals are to produce a complete design for TDS and develop a prototype system. Although limited in functionality, the prototype must be complete enough to pass an early look test based on a test plan generated by LAS. The sponsor has the ultimate authority to decide whether or not the prototype justifies further development. Phase 1 consists of the following tasks:

- **Research** - Previous work by LAS has generated many working theories and prototypes/work benches. All of this work must be analyzed further to determine which theories are best applied to TDS. In early September, when this task is scheduled to conclude, LAS will know how all of the major components of TDS will work and will have a conceptual design document that drives further development. In addition, LAS will deliver input specifications for data to be loaded into TDS.
- **Development** - Based on the outcome of the research task, LAS will develop the first limited version of TDS. Ideally prototypes developed during the research task will form the basis for this version of the system. In addition, the Linguistic team will continue to refine their research from the previous period and provide guidance to the Technical team.
- **Test Plan** - During research and development, a test plan will be developed. First, requirements will be culled from existing documentation and results from the research task. Then, these requirements will be used to develop test scripts. There are four major areas to be tested: functionality, performance, retrieval accuracy and ranking accuracy. The test plan is a deliverable required by the contract.
- **Build and Test** - A week has been reserved in the schedule to integrate the output of the development task, after which there are three weeks to execute the test, make any corrections and document the results.

While subject to change, the plan calls for the first version of TDS to contain the following features:

- A fully functional name classifier that can identify Arabic, Chinese and Hispanic names. The name classifier will be ported to C++ from LAS's already developed PC-NAS system that is currently written in Clipper.
- Name processors for Anglo and Arabic names. Note that in this phase, the Anglo name processor will not include the extended rule set for atypical Anglo names.
- A fully functional name data base with a key structure that accommodates both IPA exact match and phonetically "similar to" searching. A program to load raw data into the name data base will also be produced during this phase. The name data used will be obtained from LAS resources.
- A search engine that when given a query name and it's ethnicity will search the name data base and provide a list of matches.
- A limited version of the ranker with a sorting algorithm to be determined during development.
- A limited graphical user interface (GUI) to allow for evaluation of the TDS.

6.2 Phase 2 Development

Phase 2 provides three months to complete the development of TDS. Currently, the features to be developed in this phase are:

- Develop the Hispanic and Chinese name processors.
- Extend the Anglo name processor to include rules for atypical Anglo names and pronunciations.
- Complete the Ranker to include a sort algorithm with additional sorts as deemed useful.
- Finalize the GUI to include all features required by the sponsor and/or deemed desirable by LAS.
- Finalize the TDS documentation to include a simple user manual and descriptions of all algorithms.

Phase 2 culminates in the execution of an acceptance test with time built in for bug fixes and test documentation.

6.3 Implementation

The final task is to deliver the system to the sponsor. LAS will purchase, test and configure two high-end laptop computers, load sponsor data into TDS, provide training, and write a final report.

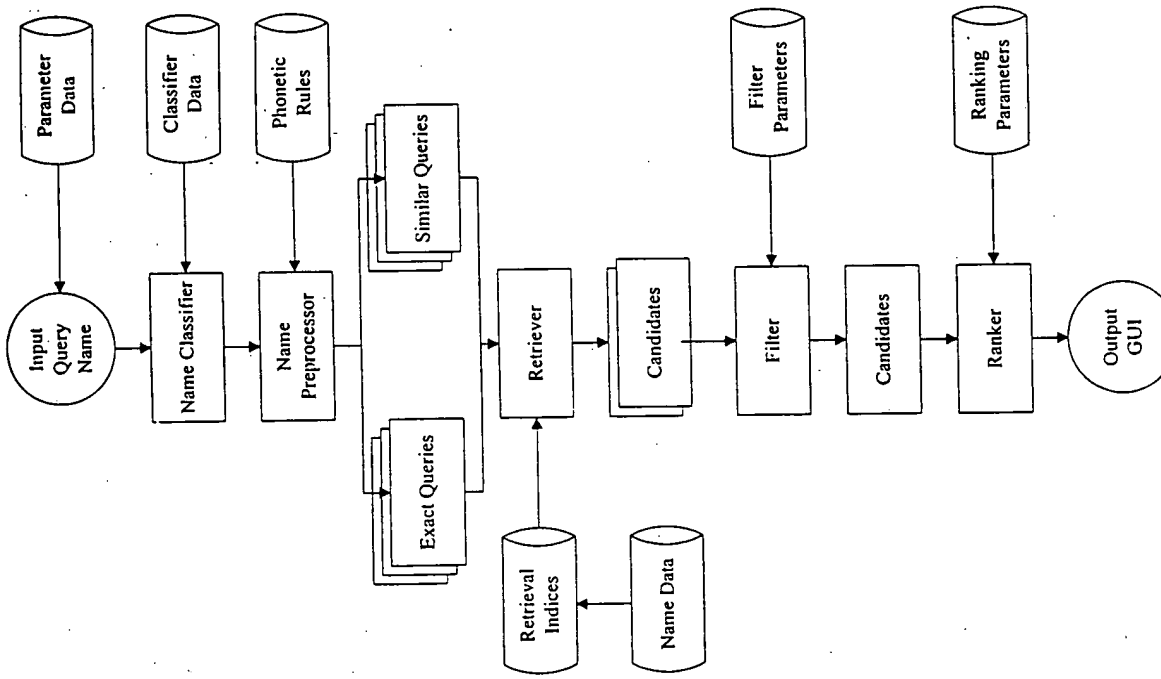
Name Search Technology Demonstration System Acceptance Test

February 11, 1998



Language Analysis Systems, Inc.
2214 Rock Hill Road—Herndon, VA—20170

**Name Search
Technology Demonstration System
(TDS)
Conceptual Design
11 February 1998**



Technology Demonstration System (TDS)

- A phonological name search system
- Technology based on the research results of an 18-month project
- A prototype developed to demonstrate the capabilities of this technology

WJ

Characteristics of TDS

- Each query drives the list of returns
 - no *a priori* set or group of names
 - “on-the-fly”, automatic process of associating query name with database names
 - allows *multiple* relationships among names
- Pronunciation and spelling contribute to measures of similarity
- System returns ranked list of similar names

WJ

Key Characteristics of the TDS

Characteristics of TDS (cont.)

- Culture-based rule sets
 - Names are automatically classified as Arabic, Mandarin Chinese, Hispanic or "Other"
 - Arabic, Mandarin Chinese and Hispanic rule sets process names based on automatic classification
 - E.g., Arabic rules: *Qaddafi ~ Khaddafi*
 - All names are processed by Anglo rules
- *fast*
- *principled*
- *fully automatic*
- *sorted returns*
- *multicultural*

WJ

WJ

Scope of the TDS

- Not a full retrieval system
 - retrieves single names (e.g., *Smith*)
- Other factors *not* covered by TDS:
 - Stems and affixes: *Vega ~ Delavega*
 - Dialect: Chinese *Ng = Wu = Huang*
 - Typographical errors: *Jpnes*
 - Perceptual issues: *Polk* misheard as *Holz*

WJ

Technical Overview of the TDS

- Written in Microsoft Visual C++ for Windows NT
- Six-month development period
- Effort concentrated on indexing strategies, search algorithms and ranker
- Minimal effort spent on User Interface
- Currently running on an IBM ThinkPad 770 with a 233 MHz Pentium MMX, 160 Mb of RAM

WJ

Requirements of the TDS

1. The TDS will incorporate a search component using phonetic name search algorithms (IPA exact match and phonetically "similar-to")
2. The TDS will incorporate a name classifier component automatically identifying a query name as a member of a specific culture for which culture-specific name processing rules can be applied
3. The cultures to be implemented in the Name Classifier component are Arabic, Chinese, Hispanic and Other (including Anglo)
4. The TDS will incorporate a rank-ordering component that ranks the results from the search component, a name database and its supporting database management system, and a graphical user interface

WJ

TDS Acceptance Test

Purpose of today's Acceptance Test:

To determine whether the TDS satisfies the requirements of the *Statement of Work*

WJ

Requirements of the TDS (cont.)

5. The individual software components that are directly related to name searching should be designed and written to be modular
6. The TDS will work on a name database consisting of at least three million names
7. The TDS will allow a user to input as a query a name in Romanized form
 - Accepts entry of a single name segment only
 - Accepts input length 2 - 30 characters as valid entry
 - Accepts lower and upper case letters as valid entry
 - Accepts alphabetic characters and apostrophe as valid entry

WJ

Requirements of the TDS (cont.)

8. Names that have been classified by the Name Classifier as Arabic, Chinese or Hispanic will be processed by their respective language-specific components, as well as by the English (Anglo)-language components. All names will be processed by the Anglo components.
9. The TDS will rank and display retrieved names in order of phonological similarity to the query name, with Exact Matches displayed at the top of an ordered list
10. The TDS will include a batch processing component
11. The ranking algorithm should be general enough to apply to the full set of names retrieved regardless of whether the technology responsible for the retrieval was IPA exact match or phonetically similar-to

WJ

Requirements of the TDS (cont.)

12. The TDS shall begin to display the results of each query against a name database of about three million names within twelve (12) seconds
13. The user shall be able to select for each query whether "similar-to" logic is to be used in the retrieval process
14. The user shall be able to select for each query whether the user will bypass the name classifier and manually specify the culture of the query name
15. The user shall be able to select for each query whether the name classifier is to be used; if not, the query shall be processed as an English (Anglo) name (in addition to the manually specified culture, if any)

LW

Requirements of the TDS (cont.)

16. The user shall be able to select for each query the maximum number of hits to be displayed
17. The reason a name was retrieved (that is, whether the retrieval of a name was due to IPA exact match or phonetically "similar-to" technology) shall be displayed along with the name
18. The TDS display will include the list of hits returned along with each name's six character group number
19. Each option selected shall become the default option and shall apply to all queries until the user changes it
20. The contractor may include other options that satisfy the needs of its developers and other personnel, as long as the default for those options is off

LW

Requirements of the TDS (cont.)

21. The TDS software shall be written with English comments embedded in the code that implements each algorithm
22. The comments shall tie blocks of code (a block of code is one or more sequential lines of code) in each algorithm's implementation to the step being implemented in the deliverable English-language narrative that describes the algorithm
23. The TDS should be designed so that new technology can be easily incorporated as it becomes available
24. The TDS will include a batch processing option for results retrieval

WJ

Requirements of the TDS (cont.)

25. The TDS will accept the following input when performing the pre-processing of the client's names database:
 - ASCII text file
 - Allowable characters include letters of the alphabet and the apostrophe (').
 - Lower case letters will be converted to upper case
 - Each record will contain two fields separated by a blank as specified below:
 - Columns 1 through 6 will contain the Group Number; the first character may be either a 'Z' or a number; the remaining 5 characters must contain a digit
 - Column 7 must be blank
 - Columns 8 through 37 will contain the name. The name must be at least 2 characters long, and no longer than 30 characters
 - Client database may exist on 3.5" diskette or CD-ROM
 - Duplicate names will be rejected

WJ

Requirements of the TDS (cont.)

26. Each query name and all the names retrieved by the TDS (not to exceed the limit in use at that time) will be saved in an internal file until the user selects to delete it
27. The TDS will also provide the capability to write the saved data onto a hit file resident on diskette(s) when directed by the user

LAJ

TDSFINAL

```
// FactorEdit.cpp : implementation file
//
// Copyright (C) 1998, Language Analysis Systems Inc.
//
#include "stdafx.h"
#include "FactorEdit.h"

#ifdef _DEBUG
#define NEW_DEBUG_NEW
#endif // _DEBUG
// #define THIS_FILE __FILE__
static char THIS_FILE[] = __FILE__;

#pragma warning(disable:4786)

// FactorEdit

IMPLEMENT_DYNAMIC_FACTORY(FactorEdit, CEdit)

FactorEdit::FactorEdit()
{
}

FactorEdit::~FactorEdit()
{
}

BEGIN_MESSAGE_MAP(FactorEdit, CEdit)
    ///((AFX_MSG_MAP(FactorEdit))
    ON_CONTROL_REFLECT(EN_KILLFOCUS, OnKillFocus)
    ON_WM_DESTROY()
    ///((AFX_MSG_MAP
END_MESSAGE_MAP()

// FactorEdit message handlers

// function to convert the text in the control to a double, and then back to a text representation
// (M-2)
void FactorEdit::OnKillFocus()
{
    ConvertToNum();
}

void FactorEdit::ConvertToNum()
{
    char numString(20 + 1);
    double aDouble;

    GetWindowText(numString, 20);
    aDouble = atof(numString);
    UpdateText("M-2", aDouble);
    SetWindowText(numString);
}

void FactorEdit::OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags)
{
    if (nChar == VK_RETURN) {
        OnKillFocus();
        return;
    }
}

CEdit::OnKeyDown(nChar, nRepCnt, nFlags);
```

[illegible]

```

end;

SetWindowProc(TDS - Name Search System - Developer Version);

// Start a thread to initialize the tdssearcher object
OsiThread
InitThread = AObjectThread(AFX_THREADPROC) searcher_init_thread_func, (LPVOID)this,
TDSBD_PRIORITY_NORMAL;

// 0, 0;
InitThreadHandle = InitThread->hThread;

return TRUE; // return TRUE unless you set the focus to a control
// BODYPART: OOI Property Pages should return FALSE

UNIT searcher_init_thread_func(LPVOID param)
{
    InitDialog *dialgr = (InitDialog *)param;
    dialgr->tdssearcherinit();
    return 0;
}

LRESULT WINAPI InitDialog::WindowProc(UINT message, WPARAM wParam, LPARAM lParam)
{
    // TODO: Add your specialized code here and/or call the base class
    if (message == TDS_INIT_DONE) {
        if (wParam == 1)
            OnOK();
        else
            OnCancel();
        return 0;
    }
    else
        return CDialog::WindowProc(message, wParam, lParam);
}

// Separate function than OnCancel to differentiate between when
// user clicks button vs when the init thread is done and posts
// a message to call OnCancel.
void InitDialog::OnCancelButton()
{
    m_statusStatic.SetWindowText("Shutting Down - Please Stand By");
    OnCancel();
}

```

```

//DC_Control (pdx, DC_VOICE_FACTOR_EDIT, a_voiceFactorEdit);
//DC_Control (pdx, DC_WAY_NAME_EDIT, a_wayNameEdit);
//DC_Control (pdx, DC_GEAR_EDIT_DIST_THRESH_EDIT, a_gearDistThresholdEdit);
//DC_Control (pdx, DC_WAY_EDIT_DIST_THRESH_EDIT, a_wayDistThresholdEdit);
//DC_Control (pdx, DC_SYLLABLE_FACTOR_EDIT, a_syllableFactorEdit);
//DC_Control (pdx, DC_EDIT_DIST_FACTOR_EDIT, a_editDistFactorEdit);
//DC_Control (pdx, DC_LEADING_CONS_FACTOR_EDIT, a_leadingConsFactorEdit);
//DC_Control (pdx, DC_FILG_NAME_EDIT, a_filgNameEdit);
//DC_Check (pdx, DC_LEFT_ALIGN_CHECK, a_leftAlign);
//DC_Check (pdx, DC_LOG_DEBUG_CHECK, a_logDebugInfo);
//)NXT_DATA_MAP
}

//BETA_MESSAGE_MAP (PamMsgDlg, CHATLOG)
//({(NXT_MSG_MAP (PamMsgDlg)
//  ON_BN_CLICKED(IDC_BUTTON_LOADFILE_BUTTON, OnLoadFileButtonClick)
//  ON_BN_CLICKED(IDC_BUTTON_EDIT_DIST_ADJUST_MORE_RADIO, OnAdjustDistMoreButtonClick)
//  ON_BN_CLICKED(IDC_BUTTON_EDIT_DIST_ADJUST_SAME_RADIO, OnAdjustDistSameButtonClick)
//  ON_BN_CLICKED(IDC_BUTTON_EDIT_DIST_ADJUST_LESS_RADIO, OnAdjustDistLessButtonClick)
//  ON_BN_CLICKED(IDC_BUTTON_EDIT_DIST_ADJUST_FILG_RADIO, OnEditDistAdjustFilgButtonClick)
//  ON_BN_CLICKED(IDC_BUTTON_EDIT_DIST_ADJUST_THRESH_RADIO, OnEditDistAdjustThresholdButtonClick)
//  ON_BN_CLICKED(IDC_BUTTON_EDIT_DIST_THRESH_RADIO, OnEditDistAdjustThresholdButtonClick)
//  )NXT_MSG_MAP
//)

//BETA_MESSAGE_MAP()
//)

//)PamMsg message handlers

void PamMsgDlg::OnClickedEditButton()
{
    m_LogFileNumEdit.GetWindowText(m_LogFileName, 1000);

    if (DeleteFile(m_LogFileName) == 0) {
        AfxMessageBox("Could Not Find The Specified File");
    }
}

void PamMsgDlg::OnOK()
{
    char nameString(100 + 1);
    bool rc = true;
    float spellingRate;
    float spellingRate;
    float cultureRate;
    float syllableRate;
    float etichRate;
    float leadingConsRate;
    float vowelRate;

    UpdateData();
    // validate the data
    // make sure we are not negative, and that at least one
    // is > 0.
    for (int i = 0; i < 7; ++i) {
        switch (i) {
            case 0:
                m_CultureFactorEdit.GetWindowText(nameString, 100);
                m_VowelRate = (float)nameString;
                if (m_CultureRate < 0.0) {
                    AfxMessageBox("The Culture Rate must be >= 0.");
                    rc = false;
                }
                break;
            case 1:
                m_EditDistFactorEdit.GetWindowText(nameString, 100);

```



```

else {
    // same for post number - except disable the pre-number radio but
    if (postNumberBPMde != TDS_BP_MXIS_MXIS) {
        m_preNumberBitDisAdjJustThresRadio.Enabled(false);
        m_preNumberBitDisAdjJustThresRadio.Enabled(false);
        m_preNumberBitDisAdjJustThresRadio.Enabled(false);
    }
    // if the postNumberBPMde is not, disable the test fields
    if (postNumberBPMde == TDS_BP_MXIS_MXIS) {
        m_testNumberLowVLTThresEdit.Enabled(false);
        m_testNumberLowVLTThresEdit.Enabled(false);
        m_testNumberHighVLTThresEdit.Enabled(false);
        m_testNumberHighVLTThresEdit.Enabled(false);
    }
    return TRUE; // return TRUE unless you set the focus to a control
    // EXCEPTION: OCX Property Pages should return FALSE
}

// UNREGISTERED: WindowProc(UINT message, WPARAM wParam, LPARAM lParam)
{
    return DialogBox(message, wParam, lParam);
}

void ParmDlg::OnBitDisAdjJustThresRadio()
{
    postNumberBPMde = TDS_BP_MXIS_MXIS;
    // enable the other radio buttons
    m_postNumberBitDisAdjJustThresRadio.Enabled(false);
    m_postNumberBitDisAdjJustThresRadio.Enabled(false);
    m_postNumberBitDisAdjJustThresRadio.Enabled(false);
}

void ParmDlg::OnBitDisAdjJustThresRadio()
{
    preNumberBPMde = TDS_BP_MXIS_MXIS;
    // disable the other radio buttons
    m_preNumberBitDisAdjJustThresRadio.Enabled(false);
    m_preNumberBitDisAdjJustThresRadio.Enabled(false);
    m_preNumberBitDisAdjJustThresRadio.Enabled(false);
}

void ParmDlg::OnBitDisAdjJustThresRadio()
{
    preNumberBPMde = TDS_BP_MXIS_THRES;
    // disable the other radio buttons
    m_preNumberBitDisAdjJustThresRadio.Enabled(false);
    m_preNumberBitDisAdjJustThresRadio.Enabled(false);
    m_preNumberBitDisAdjJustThresRadio.Enabled(false);
}

void ParmDlg::OnBitDisAdjJustThresRadio()
{
    postNumberBPMde = TDS_BP_MXIS_MXIS;
    m_preNumberBitDisAdjJustThresRadio.Enabled(true);
    m_preNumberBitDisAdjJustThresRadio.Enabled(true);
    m_preNumberBitDisAdjJustThresRadio.Enabled(true);
    m_testNumberLowVLTThresEdit.Enabled(false);
    m_testNumberLowVLTThresEdit.Enabled(false);
    m_testNumberHighVLTThresEdit.Enabled(false);
    m_testNumberHighVLTThresEdit.Enabled(false);
}

```

```

void ParmDlg::OnPreDisAdjJustThresRadio()
{
    postNumberBPMde = TDS_BP_MXIS_SINGLE;
    m_preNumberBitDisAdjJustThresRadio.Enabled(false);
    m_preNumberBitDisAdjJustThresRadio.Enabled(false);
    m_preNumberBitDisAdjJustThresRadio.Enabled(false);
    m_testNumberLowVLTThresEdit.Enabled(true);
    m_testNumberLowVLTThresEdit.Enabled(true);
    m_testNumberHighVLTThresEdit.Enabled(true);
    m_testNumberHighVLTThresEdit.Enabled(true);
}

void ParmDlg::OnPreDisAdjJustThresRadio()
{
    postNumberBPMde = TDS_BP_MXIS_THRES;
    m_preNumberBitDisAdjJustThresRadio.Enabled(false);
    m_preNumberBitDisAdjJustThresRadio.Enabled(false);
    m_preNumberBitDisAdjJustThresRadio.Enabled(false);
    m_testNumberLowVLTThresEdit.Enabled(true);
    m_testNumberLowVLTThresEdit.Enabled(true);
    m_testNumberHighVLTThresEdit.Enabled(true);
    m_testNumberHighVLTThresEdit.Enabled(true);
}

```

```
// stdafx.cpp : source file that includes just the standard includes
// TUS PCM will be the pre-compiled header
// stdafx.h will contain the pre-compiled type information
// Copyright (C) 1998, Language Analysis Systems Inc.

#include "stdafx.h"
```

```

// TDS esp : Defines the class behaviors for the application.
//
// Copyright (C) 1998, Language Analysis Systems Inc.
//
#include "ctds.h"
#include "TDS.h"
#include "TDSDlg.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

// CTDSPApp
// CTDSPApp

BEGIN_MESSAGE_MAP(CTDSPApp, CWinApp)
    //[[[MSG_MAP(CTDSPApp)
    // NOTE - the ClassWizard will add and remove mapping macros here.
    // DO NOT EDIT what you see in these blocks of generated code!
    //]]]MSG_MAP
    ON_COMMAND(ID_HELP, CWinApp::OnHelp)
END_MESSAGE_MAP()

// CTDSPApp construction
// CTDSPApp construction
CTDSPApp(CTDSPApp)
{
    // TODO: add construction code here,
    // Place all significant initialization in InitInstance
}

// The one and only CTDSPApp object
CTDSPApp theApp;

// CTDSPApp initialization
// CTDSPApp initialization

bool CTDSPApp::InitInstance()
{
    // Standard Initialization
    // If you are not using these features and wish to reduce the size
    // of your final executable, you should remove from the following
    // the specific initialization routines you do not need.

    // TODO: Place code here to handle when the dialog is
    // dismissed with OK
    else if (m_pMainWnd == NULL)
    {
        InitInstance();
        if (m_pMainWnd == NULL)
        {
            // TODO: Place code here to handle when the dialog is
            // dismissed with OK
        }
    }
    else if (m_pMainWnd == NULL)
    {
        // TODO: Place code here to handle when the dialog is
        // dismissed with OK
    }
}

```



```

    pSystem->AppendMenu(M_SEPARATOR);
    pSystem->AppendMenu(ME_STRING, IDA_ABOUTBOX, &aboutMenu);
}

// Set the icon for this dialog. The framework does this automatically
// when the application's main window is not a dialog
SetIcon(m_hIcon, TRUE); // Set big icon
SetIcon(m_hIcon, FALSE); // Set small icon

// Add the appropriate fields to the list control
// Try to create an image list and associate it with the listctrl
m_listImages.Create(116, 15, ILC_COLOR4, 0, 1);
CreateImageList();

m_listImages.Add(16m, RGB(192, 192, 192));

// Associate CImageList with CListCtrl
m_queryResultsListCtrl.SetImageList(m_listImages, LVSIL_SMALL);

// Insert columns in the list control
LV_COLUMN lvc;
lvc.mask = LVCF_FMT | LVCF_WIDTH | LVCF_TEXT;

int tempWidth;
int maxWindowWidthDialogPlayAllCols = 45;
for (unsigned int i = 0; i < sizeof col_desc / sizeof col_desc(0); i++) {
    lvc.pszText = col_desc(i).c_str();
    lvc.lParam = col_desc(i).c_str();
    // If the fields width is not specified in the
    // col_desc structure, set the width to the size of the
    // column header text.
    tempWidth = m_queryResultsListCtrl.GetColumnWidth(col_desc(i).c_str()) + 12;
    lvc.cx = tempWidth > col_desc(i).c_str().length() ? tempWidth : col_desc(i).c_str().length();
    m_queryResultsListCtrl.InsertColumn(i, lvc);
    maxWindowWidthDialogPlayAllCols += lvc.cx;
}

// Open up the log file to capture messages during the creation and
// initialization of the runner.
LogStream.OpenLogFile(16m, LOG::APP);

// Put up a splash dialog and wait until it returns. It will return if
// the user clicks on the cancel button in the splash dialog, or
// if the init thread ends (either because of an error, an successful
// initialization.
InitDialog InitDialog(NULL, 0, 0);

int initResult = InitDialog.DoModal();

// Set a pointer to the tdbsearcher object that the InitDialog
// allocated and initialized. We are responsible for deleting it.
// We need to delete it even if the init failed or was canceled, so
// that is why we make sure we assign it to our pointer variable
// regardless of the return code from DoModal.
tdbsearcher = InitDialog.getSearcher();

if (initResult == LOGS) {
    char tempString[100 + 1];

    // Set this once, here, since it will not be changing with each query

```

TOSDLG.CPP 3-24-98 12:14p

```

tdbsearcher->SetCopyVariableAddress(16m);
// Make sure the search uses our default values
SendParamToSearcher(1);

// Reflect max names to return on GUI
sprintf(tempString, "10", maxNameToReturnQuery);
m_nameToReturnEdit.SetWindowText(tempString);

// Reflect query mode on GUI
if (queryMode == TDSQ_QUERY_MODE_SIMILAR)
    CheckRadioButton(IDC_QUERY_RADIO, IDC_SIMILAR_SEARCH_RADIO, IDC_SIMILAR_SE);
else
    CheckRadioButton(IDC_QUERY_RADIO, IDC_SIMILAR_SEARCH_RADIO, IDC_QUERY_RADIO);

// Fill up the culture combo box with strings from the
// cultureStrings array
m_cultureCombo.AddString(tdbsearcher->GetCultureString(IDS_CULT_AFRICAN));
m_cultureCombo.AddString(tdbsearcher->GetCultureString(IDS_CULT_AMERICAN));
m_cultureCombo.AddString(tdbsearcher->GetCultureString(IDS_CULT_CHINESE));
m_cultureCombo.AddString(tdbsearcher->GetCultureString(IDS_CULT_HISPANIC));

// Reflect the specified culture in the culture combo box
m_cultureCombo.SelectString(-1, tdbsearcher->GetCultureString(specificCult));

// Reflect the default culture mode.
if (cultCntrMode == TDSQ_CULT_MODE_AUTO) {
    // auto classification, so select the radio button, and disable
    // the culture combo box
    CheckRadioButton(IDC_AUTO_CLASS_RADIO, IDC_SPECIFY_CLASS_RADIO, IDC_AUTO_CLASS);
    m_cultureCombo.EnableWindow(FALSE);
}
else
    CheckRadioButton(IDC_AUTO_CLASS_RADIO, IDC_SPECIFY_CLASS_RADIO, IDC_SPECIFY_CLASS);

// Call GetSystemMetrics to see how big we can make the window
int origWidth;
int origHeight;

origWidth = GetSystemMetrics(SM_CXFULLSCREEN) - 50;
origHeight = GetSystemMetrics(SM_CYFULLSCREEN) - 40;

// Don't make the window wider than is needed to display
// all the columns
origWidth = origWidth < TDSQ_MAX_SCREEN_WIDTH ? origWidth : TDSQ_MAX_SCREEN_WIDTH;
origHeight = origHeight < maxWindowWidthDialogPlayAll ? origHeight : maxWindowWidthDialogPlayAll;

SetWindowPos(hwndTop, 0, 0, origWidth, origHeight, SW_SHOWNORMAL);

enableControls(TRUE);

// Make the Settings button invisible for user versions of the app
if (def_tdsq_user_version)

```

Page 3 of 15


```

char      numString(100 * 1);
int       numHits = tdsSearcher->getNumResults();
char      logString(1000 * 1);
int       i;
char      numFmtNumString(10);

// user version uses 2 decimal precision
sprintf(TDS_USER_VERSION, "%2.4f");
strcpy(numFmtNumString, "%2.4f");
else
strcpy(numFmtNumString, "%2.2f");
endif

if (done)
{
    sprintf(numHitsString, "%d hits", numHits);
    m_queryResultListCtrl.SetItem(0, 4, numHitsString);
}

logStream << "Hits To Follow:" << endl;

memset(lvi, 0, sizeof(lvi));
lvi.mak = LVIF_TEXT | LVIF_IMAGE | LVIF_PARAM;
lvi.iImage = 2;
lvi.iSubItem = 0;
lvi.iItem = 0;

vector<EXCHName *> resultNames;

// fill up the list box with the hits we got.
if (firstWaveOfResults) { // these are exact matches.
    // Start at 1 since the header is at location 0
    lvi.iItem = 1;
    // copy the pointers to the result names (EXCHName *) to our vector
    tdsSearcher->getResultNamesForExactMatch(resultNames);
}
else { // these are results from the second phase, so
    lvi.iItem = endIndexForCurrentQuery; // start after where we left off from the first phase.
    // copy the pointers to the result names (EXCHName *) to our vector
    tdsSearcher->getResultNamesForSimilarMatch(resultNames);
}

int numNamesToAppend = resultNames.size();
char tempNameString(TDS_MAX_NAME * 10); // for indented name
strcpy(tempNameString, " "); // set first 2 chars to a space for indent
m_queryResultListCtrl.SetRedraw(false);
for (i = 0; i < numNamesToAppend; i++) {
    sprintf(numString, "%d", lvi.iItem);
    lvi.SetText = numString;
    EXCHName *exchResultName = resultNames[i];
    if (exchResultName->getIsExact()) // check image
        lvi.iImage = 3; // blank image
    else
        lvi.iImage = 2; // blank image
    m_queryResultListCtrl.InsertItem(lvi);
    // add two spaces at beginning of the name so it gets indented
    strcpy(tempNameString + 2, (LPTSTR) (exchResultName->getCtrl().c_str()));
    m_queryResultListCtrl.SetItem(0, 1, tempNameString);
    // display the group number of the name that was returned
}

```

```

else
{
    if (currentQueryStatus == TDS_QUERY_STATUS_ERROR)
    {
        m_queryResultListCtrl.SetItem(0, 4, "Error");
        done = true;
        logStream << "Query processing stopped due to an error: " << endl;
    }
    else
    {
        if ((queryPhase == 1) && (queryMode == TDS_QUERY_MODE_SIMILAR))
        {
            // we are at the end of phase 1, and we are supposed to
            // do phase 2, so update the status to that effect, and
            // start the next thread. Also include the number of hits
            // from phase 1 in the header.
            flushBankerResultListCtrl(false) /* not done */;
            true /* first wave of results */;
            queryPhase = 2;
            logStream << "Starting phase 2 of the search: " << endl;
            CtrlThread *aThread;
            aThread = AcdBeginThread((AFX_THREADPROC) similar_search_thread_func, (LPVOID) 0);
            // his.
            TDSQD_PRIORITY_NORMAL, 0, 0);
            currentRunningThreadHandle = aThread->m_hThread;
        }
        else
        {
            // we must be done.
            flushBankerResultListCtrl(true, // done
            queryPhase == 1 /* first wave of results ? */);
            enableControls(TRUE);
            done = true;
        }
    }
}

// release the resources allocated for the query if
// - there was an error
// - the user canceled
// - we are done
if (done)
{
    // update the status to say we are done
    PostMessage(TDSQD_UPDATE_STATUS_MSG, TDSQD_STATUS_SEARCH_COMPLETED, 0);
    /* ??? commented out for now
    if (*exchName != EOS)
    {
        outputWatchResults();
    }
    */
    // close the log file
    logStream << "Query has ended " << endl;
    logStream.close();
    OutDialogCtrl(m_queryNameEdit);
}

return 0;
}

void CTDSDlg::flushBankerResultListCtrl(bool done, bool firstWaveOfResults)
{
    LV_ITEM lvi;
    char numHitsString(100 * 1);
}

```



```

m_query#resultsCtrl.SetItemText(lvi.item, 2, extResultName ->getNewCode());

// display the culture of the name that was returned (e.g. which pipe)
m_query#resultsCtrl.SetItemText(lvi.item, 3,

tbdSearcher->getCultureString(extResultName ->getPipeCode()));

** true();
** true();

// overall score
print(namString, numericFormatString, extResultName ->getWeightedScore());
m_query#resultsCtrl.SetItemText(lvi.item, 4, namString);

// only the developer version gets the scores past the overall score
// instead TREC_IDE_V050104

    edit dist score
    if (rankParams.getPhonetic() > 0) {
        print(namString, numericFormatString, extResultName ->getPhoneticScore());
        m_query#resultsCtrl.SetItemText(lvi.item, 5, namString);
    }
    else
        m_query#resultsCtrl.SetItemText(lvi.item, 5, "N/A");

    // spelling 1 score
    if (rankParams.getSpelling1() > 0) {
        print(namString, numericFormatString, extResultName ->getSpellingScore());
        m_query#resultsCtrl.SetItemText(lvi.item, 6, namString);
    }
    else
        m_query#resultsCtrl.SetItemText(lvi.item, 6, "N/A");

    // spelling 2 score
    if (rankParams.getSpelling2() > 0) {
        print(namString, numericFormatString, extResultName ->getSpellingScore());
        m_query#resultsCtrl.SetItemText(lvi.item, 7, namString);
    }
    else
        m_query#resultsCtrl.SetItemText(lvi.item, 7, "N/A");

    // syllable score
    if (rankParams.getSyllable() > 0) {
        print(namString, numericFormatString, extResultName ->getSyllableScore());
        m_query#resultsCtrl.SetItemText(lvi.item, 8, namString);
    }
    else
        m_query#resultsCtrl.SetItemText(lvi.item, 8, "N/A");

    // lead core score
    if (rankParams.getLeadCore() > 0) {
        print(namString, numericFormatString, extResultName ->getLeadCoreScore());
        m_query#resultsCtrl.SetItemText(lvi.item, 9, namString);
    }
    else
        m_query#resultsCtrl.SetItemText(lvi.item, 9, "N/A");

    // vowel score
    if (rankParams.getVowel() > 0) {
        print(namString, numericFormatString, extResultName ->getVowelScore());
        m_query#resultsCtrl.SetItemText(lvi.item, 10, namString);
    }
    else
        m_query#resultsCtrl.SetItemText(lvi.item, 10, "N/A");

    // culture score
    if (rankParams.getCulture() > 0) {
        print(namString, numericFormatString, extResultName ->getCultureScore());
    }
}

```

```

// rankerHighThreshold, otherwise set it to
// rankerLowThreshold. Note that since the searcher
// has a pointer to our RankerStats object, we do not
// need to update the TDSearcher explicitly
if (queryMgr == rankerHighThreshold)
    rankerPma.setThreshold(rankerHighThreshold);
else
    rankerPma.setThreshold(rankerLowThreshold);

// now set the fat Ranker stuff unless we are not doing any post-
// ranker adjustment
if (postRankerPhase != TDS_IP_MERGE_RANKER) {
    if (queryMgr == fatRankerVfThreshold) {
        tdsSearcher.setFatRankerThreshold(fatRankerVfThreshold);
    }
    else {
        tdsSearcher.setFatRankerThreshold(fatRankerLowVfThreshold);
        tdsSearcher.setFatRankerThreshold(fatRankerMedVfThreshold);
    }
}

if (tdsSearcher.submitQuery(queryName)) {
    lv1.isItem = 0;

    // add a line for the query info into the list ctrl
    lv1.mask = LVIF_TEXT | LVIF_IMAGE;
    lv1.image = 0;
    lv1.item = 0;
    lv1.text = "";

    m_queryResultsListCtrl.InsertItem(slv1);
    m_queryResultsListCtrl.SetItemText(0, 1, queryName);

    userNotSetup = false;
    currentQueryStatus = TDS_QUERY_STATUS_OK;
    queryPhase = 1;
    m_queryResultsListCtrl.SetItemText(0, 4, "Working");
    enableControls(FALSE);

    // set the focus to the list box, since the
    // query name is now disabled. When the query is
    // completed, the query edit control will be re-enabled, and
    // the focus set back to the control.
    m_queryResultsListCtrl.SetFocus();

    // set up a query info structure for this query
    queryStats = new TDSearcher::tbl_query_stats_t;
    queryStats->clear();

    // update our query stats so that the user can see the number of
    // groups and the number of cultures searched
    tdsSearcher->getQueryStats(queryStats);

    // Add it as the extra item data to the query header in the listctrl
    m_queryResultsListCtrl.SetItemData(0, (DWORD)queryStats);

    // update the classification
    PostMessage(TDSE_UPDATE_CLASSIFICATION_MSG, 0, 0);

    OnThread *oThread;
    oThread = MxObjThread(mfx_threadFunc, exact_search_thread_func, (LPVOID)this,
                          0, 0);
    currentlyRunningThreadHandle = oThread->m_thread;
}
else {
    // _PRGILITY_KERNAL, 0, 0);
    waitCu
}

```

```

waitCursor = LoadCursor(NULL, IDC_ARROW);
SetCursor(waitCursor);

// update the GUI to say we are doing the similar to search
PostMessage(HWND_STATUS_F60, TMSG_STATUS_F60_UPDATE_PANEL_SOW, 0, 0);

// If there was a problem, set the status so that the
// error will be reported when we post the message to
// update the GUI. Also, this will prevent the second
// phase from executing.
if (tdsSearcher->searcherSimultaneous() == false)
    currentQueryStatus = TMSG_QUERY_STATUS_ERROR;

// update the status for this last phase of the search
tdsSearcher->setQueryStatus(queryStatus);

// post a message to reflect the results of the Similar Search
PostMessage(TMSG_UPDATE_QUERY_RESULTS_M82, 0, 0);

// let the rest of the app know that there is no thread running.
currentlyRunningThreadHandle = 0;

return rc;
}

void CTDSDlg::OnSize(UINT nType, int cx, int cy)
{
    DialogBox::OnSize(nType, cx, cy);

    if (m_queryResultsListCtrl.m_hWnd != NULL)
    {
        CRect pageRect;
        CRect controlRect;
        int windowWidth;

        // get the rect we have to work with
        GetClientRect(&pageRect);

        windowWidth = pageRect.right - pageRect.left;

        // make sure the results list control does not go past the bottom
        // or right edges. Leave space for the status bar.
        m_queryResultsListCtrl.GetClientRect(&controlRect);
        ScreenToClient(&controlRect);
        controlRect.left = 5;
        controlRect.right = pageRect.right - 5;
        controlRect.bottom = pageRect.bottom - 15;
        m_queryResultsListCtrl.MoveWindow(&controlRect);

        // keep the status label the bottom
        m_statusLabel.GetWindowRect(&controlRect);
        ScreenToClient(&controlRect);
        controlRect.top = pageRect.bottom - 19;
        controlRect.bottom = pageRect.bottom - 2;
        m_statusLabel.MoveWindow(&controlRect);

        // keep the status text field at the bottom and the remaining
        // width of the status bar, leaving room for the exit button.
        m_statusEdit.GetWindowRect(&controlRect);
        ScreenToClient(&controlRect);
        controlRect.right = pageRect.right - 70;
        controlRect.bottom = pageRect.bottom - 22;
        controlRect.bottom = pageRect.bottom - 2;
        m_statusEdit.MoveWindow(&controlRect);
    }

    // keep the legend static field at the bottom of the window
    m_legendStatic.GetWindowRect(&controlRect);
    ScreenToClient(&controlRect);
    controlRect.top = pageRect.bottom - 19;
    controlRect.bottom = pageRect.bottom - 2;
    m_legendStatic.MoveWindow(&controlRect);

    // keep the legend bitmap at the bottom
    m_legendBitmap.GetWindowRect(&controlRect);
    ScreenToClient(&controlRect);
    controlRect.top = pageRect.bottom - 22;
    controlRect.bottom = pageRect.bottom - 2;
    controlRect.left = pageRect.left - 60;
    controlRect.right = pageRect.right - 2;
    m_legendBitmap.MoveWindow(&controlRect);

    // keep the exit button at the bottom and to the right
    m_exitButton.GetWindowRect(&controlRect);
    ScreenToClient(&controlRect);
    controlRect.top = pageRect.bottom - 22;
    controlRect.bottom = pageRect.bottom - 2;
    controlRect.left = pageRect.left - 60;
    controlRect.right = pageRect.right - 2;
    m_exitButton.MoveWindow(&controlRect);

    // keep the clear results button at the bottom, and about
    // 2/3 the width from the left side
    m_clearResultButton.GetWindowRect(&controlRect);
    ScreenToClient(&controlRect);
    controlRect.right = (windowWidth * 2) / 3;
    controlRect.left = controlRect.right - 70;
    controlRect.top = pageRect.bottom - 55;
    controlRect.bottom = pageRect.bottom - 30;
    m_clearResultButton.MoveWindow(&controlRect);

    // keep the save results button at the bottom, and about
    // 1/3 the width from the left side
    m_saveResultButton.GetWindowRect(&controlRect);
    ScreenToClient(&controlRect);
    controlRect.left = windowWidth / 3;
    controlRect.right = controlRect.left + 70;
    controlRect.top = pageRect.bottom - 55;
    controlRect.bottom = pageRect.bottom - 30;
    m_saveResultButton.MoveWindow(&controlRect);

    }

void CTDSDlg::OnClearResultsButton()
{
    // go through and look for header items. For
    // each one we find, we must delete it's item data.
    int nItemCount;
    TDSSearcher::tds_query_status_t *queryStatus;
    nItemCount = m_queryResultsListCtrl.GetItemCount();
    for (int i = 0; i < nItemCount; i++)
    {
        queryStatus = TDSSearcher::tds_query_status_t *is_queryResultsListCtrl.GetItemData(i);
        if (queryStatus != NULL)
            delete queryStatus;
    }
    m_queryResultsListCtrl.DeleteAllItems();
}

```



```

    .. generated from query -> ld, hdYr"
    .. exact match on group -> ld, hdYr"
    .. matched exactly (phonetically) -> ld, hdYr"
    .. led by edit distance -> ld, hdYr"
    .. parsed edit distance -> ld, hdYr"
    .. areas retrieved from similar search -> ld, hdYr"
    .. areas that passed similar edit distance -> ld, hdYr"
    .. esString(MDS_CULT_MASSLO),
    .. esString(queryStatsPfr~secondCulture),
    .. yGroups[0],
    .. yGroups[1],
    .. smTheExactMatchChcGroup[0],
    .. smTheExactMatchChcGroup[1],
    .. smTheMatchedExactly[0],
    .. smTheMatchedExactly[1],
    .. pcCheckedSmTheDisac[0],
    .. pcCheckedSmTheDisac[1],
    .. pOfThePassedDisac[0],
    .. pOfThePassedDisac[1],
    .. idColumnsThatRetrievedFromSimilarSearch[0],
    .. idColumnsThatRetrievedFromSimilarSearch[1],
    .. smThePassedSimilarEditDistance[0],

```

```

        *result = 0;

        would OTSSDg::OnDemandAction()

        turnslog      parmslog(MML, krunkerParms);

        parmslog.setLogLevel(logFileLevel);
        parmslog.setLogVerbInfo(logVerbInfo);
        parmslog.setVerbName(watchName);
        parmslog.setWatchExitDlIsThresh(watchExitDlIsThresh);
        parmslog.setWatchDlIsThresh(groupDlIsThresh);

```

```

false
    *
    printf(outputline, "%c %6.4st-30.10s %6.6s %8-10.10s\n",
        outputline);
}

// narrative paragraph number 4.3.1
bool CTSIDlg::validateQueryParameters()
{
    bool rc = true;
    char nameString(100 + 1);

    // make sure max names is specified ok
    m_nadNameToGetNumEdit.GetText(nameString, 100);
    m_nadNameToGetNumQuery = atoi(nameString);
    if ((m_nadNameToGetNumQuery < 1) || (m_nadNameToGetNumQuery > 1000))
    {
        MessageBox("The max names to return value must be between 1 and 1000");
        rc = false;
    }

    if (rc)
    {
        if (m_CultMode == TCULT_MODE_SPECIFY)
        {
            if (validateCultureControl() == false)
            {
                // get the most current specification from the culture combo box
                MessageBox("The specified culture is invalid.");
                rc = false;
            }
        }
        return rc;
    }

    void CTSIDlg::OnInitialSearchRadio()
    {
        queryMode = TCSQ_QUERY_MODE_SIMILAR;
    }

    void CTSIDlg::OnExactSearchRadio()
    {
        queryMode = TCSQ_QUERY_MODE_EXACT;
    }

```

Page 11 of 15

```

    << endl;

    // Culture
    ostream << "Culture: " << (LPCWSTR)m_queryResultsCtrl.GetItemCount(1, 3);
    << endl;

    // number of hits
    ostream << "Number of Hits: " << (LPCWSTR)m_queryResultsCtrl.GetItemCount(
        0, 4) << endl;

    // only the developer version gets the full set of scores
    #ifdef TUTOR_USER_VERSION
        printf(outputLine, "%c %d.4s %-30.30s %-6.6s %-10.10s\n",
            '\t', m_hits[0], m_hits[1], m_hits[2], m_hits[3]);

        // Rank
        printf(outputLine, "%c %d.4s %-30.30s %-6.6s %-10.10s\n",
            '\t', m_hits[4], m_hits[5], m_hits[6], m_hits[7]);

        // Name
        printf(outputLine, "%c %d.4s %-30.30s %-6.6s %-10.10s\n",
            '\t', m_hits[8], m_hits[9], m_hits[10], m_hits[11]);

        // Group
        printf(outputLine, "%c %d.4s %-30.30s %-6.6s %-10.10s\n",
            '\t', m_hits[12], m_hits[13], m_hits[14], m_hits[15]);

        // Score
        printf(outputLine, "%c %d.4s %-30.30s %-6.6s %-10.10s\n",
            '\t', m_hits[16], m_hits[17], m_hits[18], m_hits[19]);

        // Rule Set
        printf(outputLine, "%c %d.4s %-30.30s %-6.6s %-10.10s\n",
            '\t', m_hits[20], m_hits[21], m_hits[22], m_hits[23]);

        // Spell 1
        printf(outputLine, "%c %d.4s %-30.30s %-6.6s %-10.10s\n",
            '\t', m_hits[24], m_hits[25], m_hits[26], m_hits[27]);

        // Spell 2
        printf(outputLine, "%c %d.4s %-30.30s %-6.6s %-10.10s\n",
            '\t', m_hits[28], m_hits[29], m_hits[30], m_hits[31]);

        // Syllable
        printf(outputLine, "%c %d.4s %-30.30s %-6.6s %-10.10s\n",
            '\t', m_hits[32], m_hits[33], m_hits[34], m_hits[35]);

        // Lead Case
        printf(outputLine, "%c %d.4s %-30.30s %-6.6s %-10.10s\n",
            '\t', m_hits[36], m_hits[37], m_hits[38], m_hits[39]);

        // Vowel
        printf(outputLine, "%c %d.4s %-30.30s %-6.6s %-10.10s\n",
            '\t', m_hits[40], m_hits[41], m_hits[42], m_hits[43]);

        // Culture
        printf(outputLine, "%c %d.4s %-30.30s %-6.6s %-10.10s\n",
            '\t', m_hits[44], m_hits[45], m_hits[46], m_hits[47]);

        // Rank
        printf(outputLine, "%c %d.4s %-30.30s %-6.6s %-10.10s\n",
            '\t', m_hits[48], m_hits[49], m_hits[50], m_hits[51]);

        // Name
        printf(outputLine, "%c %d.4s %-30.30s %-6.6s %-10.10s\n",
            '\t', m_hits[52], m_hits[53], m_hits[54], m_hits[55]);

        // Group
        printf(outputLine, "%c %d.4s %-30.30s %-6.6s %-10.10s\n",
            '\t', m_hits[56], m_hits[57], m_hits[58], m_hits[59]);

        // Score
        printf(outputLine, "%c %d.4s %-30.30s %-6.6s %-10.10s\n",
            '\t', m_hits[60], m_hits[61], m_hits[62], m_hits[63]);

        // Rule Set
        printf(outputLine, "%c %d.4s %-30.30s %-6.6s %-10.10s\n",
            '\t', m_hits[64], m_hits[65], m_hits[66], m_hits[67]);

        // Spell 1
        printf(outputLine, "%c %d.4s %-30.30s %-6.6s %-10.10s\n",
            '\t', m_hits[68], m_hits[69], m_hits[70], m_hits[71]);

        // Spell 2
        printf(outputLine, "%c %d.4s %-30.30s %-6.6s %-10.10s\n",
            '\t', m_hits[72], m_hits[73], m_hits[74], m_hits[75]);

        // Syllable
        printf(outputLine, "%c %d.4s %-30.30s %-6.6s %-10.10s\n",
            '\t', m_hits[76], m_hits[77], m_hits[78], m_hits[79]);

        // Lead Case
        printf(outputLine, "%c %d.4s %-30.30s %-6.6s %-10.10s\n",
            '\t', m_hits[80], m_hits[81], m_hits[82], m_hits[83]);

        // Vowel
        printf(outputLine, "%c %d.4s %-30.30s %-6.6s %-10.10s\n",
            '\t', m_hits[84], m_hits[85], m_hits[86], m_hits[87]);

        // Culture
        printf(outputLine, "%c %d.4s %-30.30s %-6.6s %-10.10s\n",
            '\t', m_hits[88], m_hits[89], m_hits[90], m_hits[91]);

        // Rank
        printf(outputLine, "%c %d.4s %-30.30s %-6.6s %-10.10s\n",
            '\t', m_hits[92], m_hits[93], m_hits[94], m_hits[95]);

        // Name
        printf(outputLine, "%c %d.4s %-30.30s %-6.6s %-10.10s\n",
            '\t', m_hits[96], m_hits[97], m_hits[98], m_hits[99]);

        // Group
        printf(outputLine, "%c %d.4s %-30.30s %-6.6s %-10.10s\n",
            '\t', m_hits[100], m_hits[101], m_hits[102], m_hits[103]);

        // Score
        printf(outputLine, "%c %d.4s %-30.30s %-6.6s %-10.10s\n",
            '\t', m_hits[104], m_hits[105], m_hits[106], m_hits[107]);

        // Rule Set
        printf(outputLine, "%c %d.4s %-30.30s %-6.6s %-10.10s\n",
            '\t', m_hits[108], m_hits[109], m_hits[110], m_hits[111]);

        // Spell 1
        printf(outputLine, "%c %d.4s %-30.30s %-6.6s %-10.10s\n",
            '\t', m_hits[112], m_hits[113], m_hits[114], m_hits[115]);

        // Spell 2
        printf(outputLine, "%c %d.4s %-30.30s %-6.6s %-10.10s\n",
            '\t', m_hits[116], m_hits[117], m_hits[118], m_hits[119]);

        // Syllable
        printf(outputLine, "%c %d.4s %-30.30s %-6.6s %-10.10s\n",
            '\t', m_hits[120], m_hits[121], m_hits[122], m_hits[123]);

        // Lead Case
        printf(outputLine, "%c %d.4s %-30.30s %-6.6s %-10.10s\n",
            '\t', m_hits[124], m_hits[125], m_hits[126], m_hits[127]);

        // Vowel
        printf(outputLine, "%c %d.4s %-30.30s %-6.6s %-10.10s\n",
            '\t', m_hits[128], m_hits[129], m_hits[130], m_hits[131]);

        // Culture
        printf(outputLine, "%c %d.4s %-30.30s %-6.6s %-10.10s\n",
            '\t', m_hits[132], m_hits[133], m_hits[134], m_hits[135]);

        // Rank
        printf(outputLine, "%c %d.4s %-30.30s %-6.6s %-10.10s\n",
            '\t', m_hits[136], m_hits[137], m_hits[138], m_hits[139]);

        // Name
        printf(outputLine, "%c %d.4s %-30.30s %-6.6s %-10.10s\n",
            '\t', m_hits[140], m_hits[141], m_hits[142], m_hits[143]);

        // Group
        printf(outputLine, "%c %d.4s %-30.30s %-6.6s %-10.10s\n",
            '\t', m_hits[144], m_hits[145], m_hits[146], m_hits[147]);

        // Score
        printf(outputLine, "%c %d.4s %-30.30s %-6.6s %-10.10s\n",
            '\t', m_hits[148], m_hits[149], m_hits[150], m_hits[151]);

        // Rule Set
        printf(outputLine, "%c %d.4s %-30.30s %-6.6s %-10.10s\n",
            '\t', m_hits[152], m_hits[153], m_hits[154], m_hits[155]);

        // Spell 1
        printf(outputLine, "%c %d.4s %-30.30s %-6.6s %-10.10s\n",
            '\t', m_hits[156], m_hits[157], m_hits[158], m_hits[159]);

        // Spell 2
        printf(outputLine, "%c %d.4s %-30.30s %-6.6s %-10.10s\n",
            '\t', m_hits[160], m_hits[161], m_hits[162], m_hits[163]);

        // Syllable
        printf(outputLine, "%c %d.4s %-30.30s %-6.6s %-10.10s\n",
            '\t', m_hits[164], m_hits[165], m_hits[166], m_hits[167]);

        // Lead Case
        printf(outputLine, "%c %d.4s %-30.30s %-6.6s %-10.10s\n",
            '\t', m_hits[168], m_hits[169], m_hits[170], m_hits[171]);

        // Vowel
        printf(outputLine, "%c %d.4s %-30.30s %-6.6s %-10.10s\n",
            '\t', m_hits[172], m_hits[173], m_hits[174], m_hits[175]);

        // Culture
        printf(outputLine, "%c %d.4s %-30.30s %-6.6s %-10.10s\n",
            '\t', m_hits[176], m_hits[177], m_hits[178], m_hits[179]);

        // Rank
        printf(outputLine, "%c %d.4s %-30.30s %-6.6s %-10.10s\n",
            '\t', m_hits[180], m_hits[181], m_hits[182], m_hits[183]);

        // Name
        printf(outputLine, "%c %d.4s %-30.30s %-6.6s %-10.10s\n",
            '\t', m_hits[184], m_hits[185], m_hits[186], m_hits[187]);

        // Group
        printf(outputLine, "%c %d.4s %-30.30s %-6.6s %-10.10s\n",
            '\t', m_hits[188], m_hits[189], m_hits[190], m_hits[191]);

        // Score
        printf(outputLine, "%c %d.4s %-30.30s %-6.6s %-10.10s\n",
            '\t', m_hits[192], m_hits[193], m_hits[194], m_hits[195]);

        // Rule Set
        printf(outputLine, "%c %d.4s %-30.30s %-6.6s %-10.10s\n",
            '\t', m_hits[196], m_hits[197], m_hits[198], m_hits[199]);

        // Spell 1
        printf(outputLine, "%c %d.4s %-30.30s %-6.6s %-10.10s\n",
            '\t', m_hits[200], m_hits[201], m_hits[202], m_hits[203]);

        // Spell 2
        printf(outputLine, "%c %d.4s %-30.30s %-6.6s %-10.10s\n",
            '\t', m_hits[204], m_hits[205], m_hits[206], m_hits[207]);

        // Syllable
        printf(outputLine, "%c %d.4s %-30.30s %-6.6s %-10.10s\n",
            '\t', m_hits[208], m_hits[209], m_hits[210], m_hits[211]);

        // Lead Case
        printf(outputLine, "%c %d.4s %-30.30s %-6.6s %-10.10s\n",
            '\t', m_hits[212], m_hits[213], m_hits[214], m_hits[215]);

        // Vowel
        printf(outputLine, "%c %d.4s %-30.30s %-6.6s %-10.10s\n
```

TDSDLG.CPP 3-24-98 12:14p

```

// at least 2 characters, we do not have to protect against
// a divide by zero
float
CTSDlg::Calculate(char *queryName)
{
    int len = 0;
    int numVowels = 0;
    while (*queryName)
    {
        if ((*queryName == 'A') || (*queryName == 'E') ||
            (*queryName == 'I') || (*queryName == 'O') ||
            (*queryName == 'U'))
        {
            numVowels++;
        }
        len++;
        queryName++;
    }
    return ((float)numVowels / (float)len);
}

void CTSDlg::OnListHeaderClicked(HWND hwnd, LPARAM lParam)
{
    HD_NOTIFY *pnm = (HD_NOTIFY *) lParam;

    // first make sure there is not a search in progress
    // we really should have a separate variable for this,
    // but we just check the state of the cancel button
    // to make sure it is enabled
    if (m_cancelButton.IsWindowEnabled() == FALSE)
    {
        if (pnm->button == 0)
        {
            // User clicked on header using left mouse button
            if (pnm->item == sortedCol)
            {
                isSortAscending = !isSortAscending;
            }
            else
            {
                isSortAscending = TRUE;
            }
            sortedCol = pnm->item;
            sortItems(-1);
        }
    }
    *pResult = 0;
}

typedef struct tds_listcontents_tag
{
    int batch; // which query batch is this
    char itemType; // 'A' for query header, 'B' for Bacteria, 'C' for non bacteria
    int rank;
    char name[50 + 1];
    char group[20 + 1];
    char ruleSet[20 + 1];
    char score[20 + 1];

    #ifndef TDS_USER_VERSION
    char editDisScore[20 + 1];
    char spellScore[20 + 1];
    char syllableScore[20 + 1];
    char sylLabelScore[20 + 1];
    char leadScore[20 + 1];
    char vowelScore[20 + 1];
    char cultureScore[20 + 1];
    #endif
}

// make sure that the specified culture (what is in the edit portion of the
// combo box) is one of the cultures in our cultureStrings array.
m_cultureCombo.GetWindowText(selectCol, 100);
if (!strcmp(selectCol, tdsSearcher->getCultureString(TDS_CULT_AZTEC)))
    specifiedCult = TDS_CULT_AZTEC;
}
else
{
    if (!strcmp(selectCol, tdsSearcher->getCultureString(TDS_CULT_APABIC)))
    {
        specifiedCult = TDS_CULT_APABIC;
    }
    else
    {
        if (!strcmp(selectCol, tdsSearcher->getCultureString(TDS_CULT_CHINESE)))
        {
            specifiedCult = TDS_CULT_CHINESE;
        }
        else
        {
            if (!strcmp(selectCol, tdsSearcher->getCultureString(TDS_CULT_HISPANIC)))
            {
                specifiedCult = TDS_CULT_HISPANIC;
            }
            else
            {
                specifiedCult = TDS_CULT_CHINESE;
            }
        }
    }
}

return rc;
}

void CTSDlg::SendMessageToSearcher()
{
    // now set the tdsSearcher object so that it uses these
    // new values. Note that we do not have to pass the
    // via info, since that ultimately affects the filter threshold.
    // and we want be able to compute that until we get the query name.
    tdsSearcher->setLanguageInfo(languageInfo);
    tdsSearcher->setMatchCriteria(criteria);
    tdsSearcher->setMatchThreshold(matchThreshold);
    tdsSearcher->setMatchThreshold(matchThreshold);
    tdsSearcher->setMatchThreshold(matchThreshold);
    tdsSearcher->setMatchThreshold(matchThreshold);

    // note that since we have already verified that the name is

```

```

TDSQueryer::tds_query_stats_t "queryStats;
char sortField[50] = "1";
bool isSortAscending;
} tds_listContents_t;

// sorts the items in the list control by the column specified by
// sortField. The isSortAscending variable determines if the sort should
// be ascending or descending.
// Since the list control contains results for multiple queries,
// the sort needs to preserve the clustering of results within a
// query.
// We do the sort by going through the list control and creating a structure
// for each item. Each structure is placed in an array, and the array is sorted.
// All the items are removed from the list control, and the sort structure is used
// to re-populate the list control.
// The numRows parameter allows us to specify that only the first N rows
// are to be sorted. This is useful to reduce the work required when a new
// set of results are added to the listCtrl, since the existing rows are already
// sorted. If the numRows parameter is -1, all the rows are sorted.
void TDSQ19::sortItems(int numRows)
{
    tds_listContents_t "listContentsBlock;
    int batchNum = 0;
    char itemType;
    D3_ITBN templstViewItem;
    int i;
    char numGetting[100] = "1";

    if (numRows == -1)
        numRows = m_queryResultListCtrl.GetItemCount();
    if (numRows) {
        // allocate the data block
        listContentsBlock = new tds_listContents_t(numRows);
        for (i = 0; i < numRows; i++) {
            // see if this is a new query
            listContentsBlock(i).queryStats =
                (TDSQueryer::tds_query_stats_t *)m_queryResultListCtrl.GetItem(
                    i).listContentsBlock(i).queryStats != NULL) ?
                m_queryStats : "A";
            itemType = "A";
        }
        else {
            templstViewItem.mask = LVIF_IMAGE;
            templstViewItem.isBitmap = 0;
            templstViewItem.item = i;
            m_queryResultListCtrl.GetItem(i).templstViewItem;
            if (templstViewItem.image == 3) // check mark
                itemType = "B";
            else
                itemType = "C";
        }
        listContentsBlock(i).batch = batchNum;
        listContentsBlock(i).itemType = itemType;
        listContentsBlock(i).rank = atoi((LPCTSTR)m_queryResultListCtrl.GetItem(i, 0));
        strategy(listContentsBlock(i).name);
        (LPCTSTR)m_queryResultListCtrl.GetItem(i, 0);
        listContentsBlock(i).name[50] = '\0';
        strategy(listContentsBlock(i).group);
    }
    1), 20);
}

```

TDSQ19.CPP 3-24-98 12:14p

```

== 2), 20);
strategy(listContentsBlock(i).group[20] = '\0';
(LPCTSTR)m_queryResultListCtrl.GetItem(i, 0);
== 3), 20);
listContentsBlock(i).ruleSet[20] = '\0';
strategy(listContentsBlock(i).score);
(LPCTSTR)m_queryResultListCtrl.GetItem(i, 0);
== 4), 20);
listContentsBlock(i).score[20] = '\0';
strategy(listContentsBlock(i).editListScore);
(LPCTSTR)m_queryResultListCtrl.GetItem(i, 0);
== 5), 20);
listContentsBlock(i).editListScore[20] = '\0';
strategy(listContentsBlock(i).spellScore);
(LPCTSTR)m_queryResultListCtrl.GetItem(i, 0);
== 6), 20);
listContentsBlock(i).spellScore[20] = '\0';
strategy(listContentsBlock(i).spellScore);
(LPCTSTR)m_queryResultListCtrl.GetItem(i, 0);
== 7), 20);
listContentsBlock(i).spellScore[20] = '\0';
strategy(listContentsBlock(i).spellScore);
(LPCTSTR)m_queryResultListCtrl.GetItem(i, 0);
== 8), 20);
listContentsBlock(i).spellScore[20] = '\0';
strategy(listContentsBlock(i).spellScore);
(LPCTSTR)m_queryResultListCtrl.GetItem(i, 0);
== 9), 20);
listContentsBlock(i).spellScore[20] = '\0';
strategy(listContentsBlock(i).spellScore);
(LPCTSTR)m_queryResultListCtrl.GetItem(i, 0);
== 10), 20);
listContentsBlock(i).spellScore[20] = '\0';
strategy(listContentsBlock(i).spellScore);
(LPCTSTR)m_queryResultListCtrl.GetItem(i, 0);
== 11), 20);
listContentsBlock(i).spellScore[20] = '\0';
strategy(listContentsBlock(i).spellScore);
(LPCTSTR)m_queryResultListCtrl.GetItem(i, 0);
endif

// figure out which field is the basis of the sort, and also note
// if we should be placing larger values at the top or the bottom.
// This varies between numeric and alpha values
if (sortCol == 0) {
    sprintf(listContentsBlock(i).sortField, "%4d", listContentsBlock(i).rank);
    listContentsBlock(i).isSortAscending = isSortAscending;
}
else if (sortCol == 1) {
    strategy(listContentsBlock(i).sortField, listContentsBlock(i).name);
    listContentsBlock(i).isSortAscending = isSortAscending;
}
else if (sortCol == 2) {
    strategy(listContentsBlock(i).sortField, listContentsBlock(i).group);
    listContentsBlock(i).isSortAscending = isSortAscending;
}
else if (sortCol == 3) {
    strategy(listContentsBlock(i).sortField, listContentsBlock(i).ruleSet);
    listContentsBlock(i).isSortAscending = isSortAscending;
}
else if (sortCol == 4) {
    strategy(listContentsBlock(i).sortField, listContentsBlock(i).score);
    listContentsBlock(i).isSortAscending = isSortAscending;
}
}

```

Page 13 of 15


```

void CTSDBlg::SetDraw( BOOL bDraw )
{
    if (! bDraw) {
        if (m_redrawcount == 0) {
            Dialog::SetDraw(false);
        }
    }
    else {
        if (m_redrawcount == 0) {
            Dialog::SetDraw(true);
            m_redrawcount = 0;
            Invalidate();
        }
    }
}

int ListComparesFunc( const void *arg1, const void *arg2 )
{
    int rc;

    tds_listcontents_t *item1 = (tds_listcontents_t *)arg1;
    tds_listcontents_t *item2 = (tds_listcontents_t *)arg2;

    rc = item1->batch - item2->batch;
    if (rc == 0) {
        if (item1->itemtype == 'A')
            rc = -1;
        else
            if (item2->itemtype == 'A')
                rc = 1;
            else
                if (item2->isSortAscending == true)
                    rc = strcmp(item1->sortField, item2->sortField);
                else
                    rc = strcmp(item2->sortField, item1->sortField);
    }
}

return rc;
}

/* void OVDiaCtrl::AutoSizeColumn( int col ) {
// call this after your list control is filled
SetDraw(false);
int maxcol = col < 0 ? 0 : col;
int maxcol = col < 0 ? GetColumnCount() - 1 : col;
for (col = maxcol; col <= maxcol; col++) {
    SetColumnWidth( col, LVSCW_AUTOSIZE );
    int wcl = GetColumnWidth( col );
    SetColumnWidth( col, LVSCW_AUTOSIZE_USEHEADER );
    int wc2 = GetColumnWidth( col );
    int wc = MAX( MIN( wcl, wc2 ), MAX( wcl, wc2 ) );
    SetColumnWidth( col, wc );
}
// RecalcHeaderTips(); *** uncomment this if you use my header tip method
SetDraw(true);
// Invalidate(); *** uncomment this if you don't use my SetDraw function
}
*/

// If they hit the escape key, make sure they really want to
// exit by putting up a message box.
BOX CTSDBlg::PreTranslateMessage(MSG* pMsg)
{

```

```

    if (pMsg->message == WM_KEYDOWN) {
        if (pMsg->wparam == VK_ESCAPE) {
            // bug out of processing
            if (AfxMessageBox("Do you really want to exit", MB_YESNO) != IDYES)
                return TRUE;
        }
    }
    return Dialog::PreTranslateMessage(pMsg);
}

// this function gets called when the user clicks
void CTSDBlg::OnCancel()
{
    waitForThreadToFinish();
    // need to do this so the space allocated for the query info
    // gets deleted.
    ClearQueryInfo();
    Dialog::OnCancel();
}

```



```
// Copyright (C) 1998, Language Analysis Systems Inc.

//
//
//((AFX_INSERT_LOCATION))
// Microsoft Developer Studio will insert additional declarations immediately before the previous line.

#endif // defined(AFX_INTDIALOG_H_869B1901_986C_11D1_9552_004005115BF7_INCLUDED_)

};

//if _MSC_VER == 1000
#pragma once
#endif // _MSC_VER == 1000
// InitDialog.h : header file
//
#include <iostream>
#include "TDSYSTEM.H"
////////////////////////////////////
// InitDialog dialog
class InitDialog : public CDialog
{
// Construction
public:
    InitDialog(CWnd* pParent = NULL, int nStream = "LogStream");
    TDSearcher * m_pSearcher; // return tdsSearcher();
    bool m_bSearcherInit();

// Dialog Data
//{{AFX_DATA(InitDialog)
enum { IDD = IDD_INIT_DIALOG };
static CString s_m_sStatic;
//}}AFX_DATA

// Overrides
// ClassWizard generated virtual function overrides
//{{AFX_VIRTUAL(InitDialog)
protected:
    virtual void DoDataExchange(CDataExchange* pDX); // DDX/DDV support
    virtual BOOL OnInitDialog();
    DECLARE_MESSAGE_MAP()
//}}AFX_VIRTUAL

// Implementation
protected:
    // Generated message map functions
//{{AFX_MSG(InitDialog)
virtual BOOL OnInitDialog();
afx_msg void OnCancel();
//}}AFX_MSG
DECLARE_MESSAGE_MAP()

private:
    CDataStream *m_LogStream;
    // the searcher object
    TDSearcher ** m_pSearcher;

public:
    HANDLE m_hThreadHandle;
    bool m_bUserCancelled;
}
```



```

//(((NO_DEPENDENCIES)))
// Microsoft Developer Studio generated include file.
// Use by TUG.rc
//
// Copyright (C) 1998, Language Analysis Systems Inc.

#define IDM_ASYNBOX 0x0010
#define IDM_ASYNBOX 100
#define IDM_ASYNBOX 101
#define IDM_TUG_DIALOG 102
#define IDM_MAINFRAME 128
#define IDM_BITMAP1 129
#define IDM_LISTVIEW_BITMAP 130
#define IDM_TUG_DIALOG 131
#define IDM_BITMAP1 132
#define IDM_BITMAP1 133
#define IDC_GROUP_BOX_EDIT 1000
#define IDC_GROUP_BOX_EDIT 1001
#define IDC_GROUP_BOX_EDIT 1002
#define IDC_GROUP_BOX_EDIT 1003
#define IDC_GROUP_BOX_EDIT 1004
#define IDC_GROUP_BOX_EDIT 1005
#define IDC_GROUP_BOX_EDIT 1006
#define IDC_GROUP_BOX_EDIT 1007
#define IDC_GROUP_BOX_EDIT 1008
#define IDC_GROUP_BOX_EDIT 1009
#define IDC_GROUP_BOX_EDIT 1010
#define IDC_GROUP_BOX_EDIT 1011
#define IDC_GROUP_BOX_EDIT 1012
#define IDC_GROUP_BOX_EDIT 1013
#define IDC_GROUP_BOX_EDIT 1014
#define IDC_GROUP_BOX_EDIT 1015
#define IDC_GROUP_BOX_EDIT 1016
#define IDC_GROUP_BOX_EDIT 1017
#define IDC_GROUP_BOX_EDIT 1018
#define IDC_GROUP_BOX_EDIT 1019
#define IDC_GROUP_BOX_EDIT 1020
#define IDC_GROUP_BOX_EDIT 1021
#define IDC_GROUP_BOX_EDIT 1022
#define IDC_GROUP_BOX_EDIT 1023
#define IDC_GROUP_BOX_EDIT 1024
#define IDC_GROUP_BOX_EDIT 1025
#define IDC_GROUP_BOX_EDIT 1026
#define IDC_GROUP_BOX_EDIT 1027
#define IDC_GROUP_BOX_EDIT 1028
#define IDC_GROUP_BOX_EDIT 1029
#define IDC_GROUP_BOX_EDIT 1030
#define IDC_GROUP_BOX_EDIT 1031
#define IDC_GROUP_BOX_EDIT 1032
#define IDC_GROUP_BOX_EDIT 1033
#define IDC_GROUP_BOX_EDIT 1034
#define IDC_GROUP_BOX_EDIT 1035
#define IDC_GROUP_BOX_EDIT 1036
#define IDC_GROUP_BOX_EDIT 1037
#define IDC_GROUP_BOX_EDIT 1038
#define IDC_GROUP_BOX_EDIT 1039

```

```

#define IDC_BROWSE_BUTTON 3146
#define IDC_STATIC_BUTTON 3147
// Next default values for new objects
//
// ID#s:
#define IDC_STATIC_BUTTON 3148
#define IDC_STATIC_BUTTON 3149
#define IDC_STATIC_BUTTON 3150
#define IDC_STATIC_BUTTON 3151
#define IDC_STATIC_BUTTON 3152
#define IDC_STATIC_BUTTON 3153
#define IDC_STATIC_BUTTON 3154
#define IDC_STATIC_BUTTON 3155

```



```
// cdb_rank.h : main header file for the TDS_RANK application
//
#ifdef _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000

#include "stdafx.h"
before including this file for PCH

// main symbols
// CDB_RANK App
// See cdb_rank.cpp for the implementation of this class
//
class CDB_RANKApp : public CWinApp
{
public:
    CDB_RANKApp();

// Overrides
// ClassWizard generated virtual function overrides
//{{AFX_VIRTUAL(CDB_RANKApp)
public:
    virtual BOOL InitInstance();
//}}AFX_VIRTUAL

// Implementation

//{{AFX_MSG(CDB_RANKApp)
// NOTE - the ClassWizard will add and remove member functions here.
// DO NOT EDIT what you see in these blocks of generated code !
//{{AFX_PRIVATE_DECLS
DECLARE_MESSAGE_MAP()
}}AFX_PRIVATE_DECLS

// Microsoft Developer Studio will insert additional declarations immediately before the previous line.
//{{AFX_INSERT_LOCATION}}
// Microsoft Developer Studio will insert additional declarations immediately before the previous line.
}}
```

```

// TDSDialog.h : header file
//
// Copyright (C) 1998, Language Analysis Systems Inc.
//
// Define how wide the main dialog needs to be to display all
// the 11 rows columns without having to scroll.
//
// Define TDSO_MAX_SCREEN_WIDTH
//
class CTDSDlg : public Dialog
{
// Construction
public:
    CTDSDlg(CWnd* pParent = NULL); // standard constructor
    CTDSDlg();

    bool        deSelectSearch();
    bool        deSelectSearch();

// Dialog Data
    //AFX_DATA(CTDSDlg)
    enum { IDD = IDD_TDS_DIALOG };
    static m_ legendStatic;
    CStatic m_ legendBitmap;
    CButton m_ editButton;
    CButton m_ autoClassRadio;
    CButton m_ specifyAllRadio;
    CButton m_ exactSearchRadio;
    CButton m_ similarSearchRadio;
    CStatic m_ statusLabel;
    CComboBox m_ cultureCombo;
    CButton m_ saveResultsButton;
    CButton m_ parmButton;
    CStatic m_ statusEdit;
    CButton m_ cancelButton;
    CButton m_ queryButton;
    CButton m_ clearResultsButton;
    CListCtrl m_ queryResultsCtrl;
    CEdit m_ queryNameEdit;
    CEdit m_ autoEnterToEnterEdit;
    //AFX_DATA

    bool        deSelectGroupSearch();
    bool        deSelectGroupSearch();

    void        SetRedraw( BOOL bRedraw );

// ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CTDSDlg)
    protected:
    virtual void OnInitialDialog();
    virtual void OnDataExchange( DataExchange* pDC );
    virtual LRESULT WindowProc( UINT message, WPARAM wParam, LPARAM lParam );
    //}}AFX_VIRTUAL

// Implementation
protected:
    HICON m_hIcon;
    CDialog m_listDialog;

    // Generated message map functions
    //{{AFX_MSG(CTDSDlg)
    virtual BOOL OnInitDialog();
    afx_msg void OnSysCommand( UINT nID, LPARAM lParam );
    afx_msg HCURSOR OnQueryDragIcon();
    afx_msg void OnSize( UINT nType, int cx, int cy );
    //}}

```

```

// TDSDialog.h : header file
//
// Copyright (C) 1998, Language Analysis Systems Inc.
//
// Define how wide the main dialog needs to be to display all
// the 11 rows columns without having to scroll.
//
// Define TDSO_MAX_SCREEN_WIDTH
//
class CTDSDlg : public Dialog
{
// Construction
public:
    CTDSDlg(CWnd* pParent = NULL); // standard constructor
    CTDSDlg();

    bool        deSelectSearch();
    bool        deSelectSearch();

// Dialog Data
    //AFX_DATA(CTDSDlg)
    enum { IDD = IDD_TDS_DIALOG };
    static m_ legendStatic;
    CStatic m_ legendBitmap;
    CButton m_ editButton;
    CButton m_ autoClassRadio;
    CButton m_ specifyAllRadio;
    CButton m_ exactSearchRadio;
    CButton m_ similarSearchRadio;
    CStatic m_ statusLabel;
    CComboBox m_ cultureCombo;
    CButton m_ saveResultsButton;
    CButton m_ parmButton;
    CStatic m_ statusEdit;
    CButton m_ cancelButton;
    CButton m_ queryButton;
    CButton m_ clearResultsButton;
    CListCtrl m_ queryResultsCtrl;
    CEdit m_ queryNameEdit;
    CEdit m_ autoEnterToEnterEdit;
    //AFX_DATA

    bool        deSelectGroupSearch();
    bool        deSelectGroupSearch();

    void        SetRedraw( BOOL bRedraw );

// ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CTDSDlg)
    protected:
    virtual void OnInitialDialog();
    virtual void OnDataExchange( DataExchange* pDC );
    virtual LRESULT WindowProc( UINT message, WPARAM wParam, LPARAM lParam );
    //}}AFX_VIRTUAL

// Implementation
protected:
    HICON m_hIcon;
    CDialog m_listDialog;

    // Generated message map functions
    //{{AFX_MSG(CTDSDlg)
    virtual BOOL OnInitDialog();
    afx_msg void OnSysCommand( UINT nID, LPARAM lParam );
    afx_msg HCURSOR OnQueryDragIcon();
    afx_msg void OnSize( UINT nType, int cx, int cy );
    //}}

```

```

afx_msg void OnClickResultButton();
afx_msg void OnClickQueryResultListView(INDEX_PARAM, RESULT_PARAM);
afx_msg void OnSaveResultButton();
afx_msg void OnSaveResultButton();
afx_msg void OnSimilarSearchRadio();
afx_msg void OnExactSearchRadio();
afx_msg void OnAutoClassRadio();
afx_msg void OnSpecifyClassRadio();
virtual void OnClick();
virtual void OnCancel();
afx_msg void OnListViewHeaderClicked(INDEX_PARAM, RESULT_PARAM);
///AFX_MSGS
DECLARE_MESSAGE_MAP()

//RESULT updated for current query(int statusId);
//RESULT reflected for current query();
//RESULT update culture current query();
void enableControls(BOOL enable);
void deQuery();
void flushSearchResultCtrl(bool done, bool firstNewOfResults);
bool validateQueryParameters();
bool validateCultureControl();
void sendParamToSearcher();
float calcDiff(char *queryName);
void sortItems(int nItems);
void waitForeverToFinish();
// the searcher object
TDSearcher
** atcher;

// stuff to control the status of the current query
int
// results begin.
// undIndexForCurrentQuery; // where in listbox does current query's
// results begin.
int
// queryPhase; // which phase is currently being worked on
bool
// userWantsToStop;
int
// currentQueryStatus;
TDSearcher::tds_query_stats_t *queryStats; // struct for info about current query,
// info about the query

// make the queryName bigger than the max we will allow so
// that an explicit error message will be displayed if the
// name is too long.
char
// main screen query parameters
int
// maxLenOfReturnQuery;
int
// queryMax;
// e_tds_cultureable
// e_tds_cultureable
// e_tds_cultureable
// e_tds_culture
// e_tds_culture
// ch one did they specify?

```

TDSOLG.H 3/24/98 12:14p

```

// param dialog stuff
Parameters paramName;
// log file info
bool logDebugInfo;
char logFileName(1000 + 1);
ofstream logStream;
// thresholds for edit distances
float nameEditDistThresh;
float groupEditDistThresh;
float rankerLowVJThreshold;
float rankerHighVJThreshold;
float rankerVJThreshold;
// fat ranker info
float fatRankerLowVJThreshold;
float fatRankerHighVJThreshold;
float fatRankerVJThreshold;
float fatRankerEditDistThresh;
// e_tds_modes
e_tds_modes paramSearchMode;
// some variables to support tracing/debugging
// for the watch name
char watchName(100 + 1);
// used to monitor which column is sorted, and the direction of
// the sort.
int sortedCol;
bool isSortAscending;
int n_retrieveCount;
HANDLE currentlyRunningThreadHandle;
};

//((AFX_INSERT_LOCATION))
// Microsoft Developer Studio will insert additional declarations immediately before the previous line.
#endif // defined(AFX_TDSOLG_H__JEE210C8_3A8C_11D1_9548_004005119B7__INCLUDED_)

```

Page 2 of 2

TDS_Preprocessor

```

// Copyright (C) 1998, Language Analysis Systems Inc.
//
// approx.cpp: implementation of the approx class.
//
// See my notes in the approx.h file for what I did to this
// class (last)
//
#include "stdafx.h"
#include <iostream>
#include "approx.h"
using namespace std;

#ifdef _DEBUG
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#pragma new DBGDBG_16M
#endif

// Construction/Destruction
//
// read a file of feature differences
const bool Approx::set_distances(const char *fname)
{
    ifstream f_in(fname);
    if (!f_in.good())
        return false;

    int i, j;
    char buf[BUPSIZ]; // BUPSIZ is apparently defined in <stdio.h>

    // first, get exact matches
    // there are 15 features and the file has the number of feature differences
    // multiplied by 10
    for (i = 0; i < 256; i++)
        m_fd_matrix[i][i] = static_cast<int>(i * 15 * 10);

    // now get the feature distances from the given file
    while (f_in.getline(buf, BUPSIZ))
    {
        char *p = strchr(buf, ':');
        if (!p || !p[1])
            continue;
        if (strchr(buf, '\n'))
            continue;
        byte n = atoi(buf + 4);
        if (n < 1)
            continue;
        m_fd_matrix[static_cast<int>(buf[0])][static_cast<int>(buf[2])] = n;
    }

    f_in.close();
    return true;
}

const bool Approx::set_float_distances(const char *fname)
{
    unsigned int x, y;
    ifstream f_in(fname);
    if (!f_in.good())
        return false;
}

```

APPROX.CPP 3-24-98 11:23a

```

int i, j;
unsigned char buf[BUPSIZ]; // BUPSIZ is apparently defined in <stdio.h>

// first, get exact matches
// there are 15 features and the file has the number of feature differences
// multiplied by 10
for (i = 0; i < 256; i++)
    m_fd_matrix[i][i] = (i * 15 * 10);

while (f_in.getline(char*buf, BUPSIZ))
{
    float n = atof(const char*buf + 4);
    if (n < 1)
        continue;
    x = buf[0];
    y = buf[2];
    m_fd_matrix[x][y] = n;
}

f_in.close();
return true;

// use this to read in a file containing rec codes and their
// associated distance scores
const bool Approx::set_rec_distances(const char *filename)
{
    ifstream infile(filename);
    if (!infile.good())
        return false;

    //int i, j;
    //char buf[BUPSIZ]; // BUPSIZ is apparently defined in <stdio.h>

    infile.close();
    return true;
}

// here follows the code for the actual edit distance algo
// with several adaptations for testing purposes
//
// plain implementation of the edit distance algorithm
// this version does not use feature distance tables for
// anything.

const int Approx::plain_edit_distance(const unsigned char *query,
                                     const unsigned char *variant,
                                     float score)
{
    int rc = plain_diff(query, variant);
    score = get_plain_score(rc);
    return rc;
}

float Approx::get_plain_score()
{
    return 1.0 - m_diff / static_cast<float>(_max(m_act1_len, m_act2_len));
}

```

Page 1 of 6


```
// // approx_diff
//
//
//
//
// value using implementation
// these functions handle character comparison using
// the feature values as the actual values returned
const float Chprox::float_differences(const unsigned char *scr1,
                                     const unsigned char *scr2,
                                     float score)
{
    float rc = approx_float_diff(scr1, scr2);
    score = get_rec_gen_float_score(t);
    return rc;
}

// this version is used for generating the rec comparison scores
// it needs to call a different version of the score function
const float Chprox::rec_gen_float_differences(const unsigned char *scr1,
                                             const unsigned char *scr2,
                                             float score)
{
    float rc = approx_float_diff(scr1, scr2);
    score = get_rec_gen_float_score(t);
    return rc;
}

// here this version uses the values in the m_d_matrix
const float Chprox::approx_float_diff(const unsigned char *scr1,
                                       const unsigned char *scr2)
{
    unsigned char p_char = NULL;
    unsigned char t_char = NULL;
    float d = 0.0; float dt = 0.0; float ds = 0.0;
    int pi = 0; int ti = 0; // string indexes
    int p_x = 0; int t_x = 0; // difference-array indexes
    float lowest = 0.0;

    for (pi = 0; (p_char = scr1[pi]) != EOS; pi++)
        for (ti = 0; (t_char = scr2[ti]) != EOS; ti++)
        {
            p_x = pi + 1;
            t_x = ti + 1;
            // Pick the lowest score from the rules
            // -- upper left
            lowest = m_diff_float_array[p_x - 1][t_x - 1] + char_float_sq((unsigned int)p_char, (unsigned int)t_char);
            // -- to the left
            if (d < lowest) lowest = d;
            if (dt < lowest) lowest = dt;
            // -- above
            d = m_diff_float_array[p_x][t_x - 1];
            if (d < lowest) lowest = d;
            // -- transposition
            if (pt >= 1 && tt >= 1)
                dt = char_float_sq((unsigned int)scr1[pi - 1], (unsigned int)scr2[ti]);
        }
}
```


Page 4 of 6

11-273a 3-24-08

```

// -- above
d = m_rec_diff_float_array(p_x - 1)(t_x) + 1;
// -- upper left
lowest = m_rec_diff_float_array(p_x - 1)(t_x - 1) + rec_float_eq(p_char, t_char, diff_
** matrix);
// -- to the left
d = m_rec_diff_float_array(p_x)(t_x - 1) + 1;
if (d < lowest) lowest = d;
// -- above
d = m_rec_diff_float_array(p_x - 1)(t_x) + 1;
d2 = m_rec_diff_float_array(p_x - 1)(t_x - 1) + 1;
if (d < lowest) lowest = d;
// -- transposition
if (p1 > 1 && t1 > 1)
{
d1 = rec_float_eq(arr1(p1 - 1), arr2(t1), diff_matrix) ? 0 : 1;
d2 = rec_float_eq(arr1(p1), arr2(t1 - 1), diff_matrix) ? 0 : 1;
if (d1 == 0 && d2 == 0.0)
{
d = m_rec_diff_float_array(p_x - 2)(t_x - 2) + d1 + d2 + 1;
if (d < lowest)
lowest = d;
}
}
m_rec_diff_float_array(p_x)(t_x) = lowest;
}
m_str1_len = p1;
m_str2_len = t1;
m_rec_float_diff = get_rec_float_difference();
return m_rec_float_diff;
}
//
//
//
byte *Approx::get_fd_matrix()
{
return m_fd_matrix[0][0];
}
//
// This is only used when the threshold of the original implementation
// is set to 0
const int Approx::exact_diff(const unsigned char *scri,
const unsigned char *str2)
{
m_diff = 0;
const unsigned char *p;
const unsigned char *q;
for (p = scri, q = str2; *p != EOS && *q != EOS; p++, q++)
if (!char_eq(*p, *q))
m_diff++;
m_diff += strlen(const char *)q;
else
m_diff += strlen(const char *)p;
return m_diff;
}
//
//
//
ofstream f_out("x.out", ios::app);
if (!f_out.is_open())
return false;
f_out << scri << " " << str2 << endl << endl;
f_out << " " << " " << endl;
for (t1 = 0; t1 < m_str2_len; t1++)
f_out << str2[t1] << " ";
f_out << endl;
CString out_str;
for (p1 = 0; p1 < m_str1_len; p1++)
{
if (p1 > 1) f_out << scri[p1 - 1] << " "; else f_out << " ";
}
}

```

```

// Pick the lowest score from the rules
// -- upper left
lowest = m_rec_diff_float_array(p_x - 1)(t_x - 1) + rec_float_eq(p_char, t_char, diff_
** matrix);
// -- to the left
d = m_rec_diff_float_array(p_x)(t_x - 1) + 1;
if (d < lowest) lowest = d;
// -- above
d = m_rec_diff_float_array(p_x - 1)(t_x) + 1;
d2 = m_rec_diff_float_array(p_x - 1)(t_x - 1) + 1;
if (d < lowest) lowest = d;
// -- transposition
if (p1 > 1 && t1 > 1)
{
d1 = rec_float_eq(arr1(p1 - 1), arr2(t1), diff_matrix) ? 0 : 1;
d2 = rec_float_eq(arr1(p1), arr2(t1 - 1), diff_matrix) ? 0 : 1;
if (d1 == 0 && d2 == 0.0)
{
d = m_rec_diff_float_array(p_x - 2)(t_x - 2) + d1 + d2 + 1;
if (d < lowest)
lowest = d;
}
}
m_rec_diff_float_array(p_x)(t_x) = lowest;
}
m_str1_len = p1;
m_str2_len = t1;
m_rec_float_diff = get_rec_float_difference();
return m_rec_float_diff;
}
//
//
//
const float Approx::rec_float_distances(const unsigned char *recArray1,
const unsigned char *recArray2,
const unsigned char *recArray3(256)(256),
float score)
{
float rc = Approx::rec_float_diff(recArray1, recArray2, recArray3);
score = get_rec_float_score();
return rc;
}
const float Approx::approx_rec_float_diff(const unsigned char *arr1,
const unsigned char *arr2,
const unsigned char diff_matrix(256)(256))
{
float d, d1, d2;
int p1, t1; // string indexes
int p_x, t_x; // difference-array indexes
float lowest;
for (p1 = 0; p1 < arr1(p1) != EOS; p1++) // EOS is defined as '\0'
{
for (t1 = 0; t1 < arr2(t1) != EOS; t1++)
{
p_x = p1 + 1;
t_x = t1 + 1;
// Pick the lowest score from the rules
// -- upper left
lowest = m_rec_diff_float_array(p_x - 1)(t_x - 1) + rec_float_eq(p_char, t_char, diff_
** matrix);
// -- to the left
d = m_rec_diff_float_array(p_x)(t_x - 1) + 1;
if (d < lowest) lowest = d;
}
}
}

```

```

for (ti = 0; ti <= m_str2_len; ti++) {
    out_str.Format("%d ", m_diff_array[ti]);
    f_out << LPCSTR(out_str);
}
f_out << endl;
}
f_out << endl;
/*
for (pi = 0; pi <= m_str1_len; pi++) {
    if (pi >= 1) f_out << endl;
    for (ti = 0; ti <= m_str2_len; ti++) {
        out_str.Format("%d ", m_diff[ti]);
        f_out << LPCSTR(out_str);
    }
    f_out << endl;
}
f_out << endl;
f_out.Close();
*/
f_out.Close();
#endif

```

[illegible]

```

    m_group_array[index] = value; // assign the value
}

typedef jmp_buf_t, jmp_buf; // read any junk this left, usually comments
}

fclose(fp);
return true;
}

// =====
// what follows is a bunch of ui stuff for the file
// selection buttons, the browse function has been parametrized
// and the button buttons call it with various arguments to control
// what type of dialog will result
// =====
void CConvWin20p::Browse(CEdit *edit_box, bool state, char *type)
{
    CFileDialog *fdialog; // see the docs on CFileDialog for these parameters
    type;
    NULL;
    ON_BROWSE | ON_HIGHPRIORITY;
    Variant files (*.src); var(group_files (*.grp)|*.grp|All files (*.*)|*.|);
    NULL;

    if (t.DM_OPEN() == IDOK)
    {
        edit_box->SetWindowText("");
        edit_box->SetWindowText(t.GetPathName());
    }
}

// this selects the source file for generating groups
void CConvWin20p::OnBrowse()
{
    Browse(m_source_file_for_groups_box, true, "var");
}

// this selects the destination file for generating groups
void CConvWin20p::OnBrowse2()
{
    Browse(m_dest_file_for_groups_box, false, "grp");
}

void CConvWin20p::OnDeDupGroupsDestFileBtn()
{
    Browse(m_deDup_group_dest_box, false, "grp");
}

void CConvWin20p::OnDeDupGroupsSourceBtn()
{
    Browse(m_deDup_group_source_box, true, "grp");
}

void CConvWin20p::OnSortGroupsDestFileBtn()
{
    Browse(m_sort_groups_dest_box, false, "grp");
}

void CConvWin20p::OnSortGroupsSourceFileBtn()
{
    Browse(m_sort_groups_source_box, true, "grp");
}

```

```

void CConvWin20p::OnGetIndexesSourceBtn()
{
    Browse(m_gen_indexes_source_box, true, "grp");
}

void CConvWin20p::OnGetIndexesDestFileBtn()
{
    Browse(m_gen_indexes_idc_box, false, "idc");
}

void CConvWin20p::OnGetIndexesVecFileBtn()
{
    Browse(m_gen_indexes_vec_box, false, "vec");
}

void CConvWin20p::OnAutoGenSourceBtn()
{
    Browse(m_auto_gen_source_file_box, true, "var");
}

// =====
// this function automates the file name setting process for the intermediate
// files. This is only done in the UI, the rest of the program reads from the
// edit boxes to get the needed information.
// this function will not work if the path contains a period that is not in
// the final file name. For example if the path is d:\foo\bar\baz\file.txt
// then the functions produces erroneous results because of the directory
// named bar\baz
void CConvWin20p::OnSetFileNames()
{
    CString source;
    CString buff;

    m_auto_gen_source_file_box.GetWindowText(source);

    // translate to groups edit boxes
    m_source_file_for_groups_box.SetWindowText(source);
    source = source.SpanExcluding(".");
    buff = source + ".grp";
    m_dest_file_for_groups_box.SetWindowText(buff);

    // sort the groups edit boxes
    m_sort_groups_source_box.SetWindowText(buff);
    buff = source + ".src.grp";
    m_sort_groups_dest_box.SetWindowText(buff);

    // dedupe the groups edit boxes
    m_deDup_group_source_box.SetWindowText(buff);
    buff = source + ".ded.grp";
    m_deDup_group_dest_box.SetWindowText(buff);

    // generate the indexes edit boxes
    m_gen_indexes_source_box.SetWindowText(buff);
    buff = source + ".idc";
    m_gen_indexes_idc_box.SetWindowText(buff);
    buff = source + ".vec";
    m_gen_indexes_vec_box.SetWindowText(buff);
}

```

```

char cmdline[256] = {0};
sprintf(cmdline, "psort.com %Q %s /+1:10 %s", (LPCWSTR)short_source, (LPCWSTR)short_destination);

//PROCESS INFORMATION pi;
STARTUPINFO si;

memset(&si, 0, sizeof(STARTUPINFO));
si.cb = sizeof(STARTUPINFO);
si.dwFlags = STARTF_USESHOWWINDOW;
si.wShowWindow = SW_HIDE;
//si.wShowWindow = SW_SHOWDEFAULT;

// ok, there is a problem with the way winsp and winnt
// handle the createprocess called module when it terminates
// I have to figure out what it is
// until then the spawn version of the call is used
//
status = CreateProcess(NULL, // application name string
cmdline, // command line string
NULL, // process attributes
NULL, // thread attributes
false, // inheritance handles flag
NORMAL_PRIORITY_CLASS, // creation flags
NULL, // environment
NULL, // current directory name
&si, // startup info
4091); // process information
//
status = _spawnl(_P_WAIT,
"psort.com",
"psort.com",
"/Q",
(LPCWSTR)short_source,
"/+1:10",
(LPCWSTR)short_destination,
NULL);

EndWaitCursor();

if(status != 0) // this is for spawn
//if(status == 0) // this is for createprocess
{
    AfxMessageBox("Error sorting the group file");
    return;
}

if(!m_auto_flag == false)
    AfxMessageBox("Finished sorting groups.");

return;

// do progress bar stuff
m_progress bar.SetStep(1);
m_progress bar.SetRange(0, 100);
m_progress bar.SetPos(0);

bool status;
CString source, destination;

// this is the desktop groups button handler
void CWinVar2Sp::OnProcess()
{
    CString source, destination;
    bool status;

    // do progress bar stuff
    m_progress bar.SetStep(1);
    m_progress bar.SetRange(0, 100);
    m_progress bar.SetPos(0);
}

```

```

// do the actual translation from IPA name variant to encoded form
// narrative paragraph number 3.2.2
status = Groupify(source, destination, log);
}

if(status == false)
    MessageBox("A problem occurred while converting to groups");
else
    MessageBox("Finished converting to groups.");

return;

// get rid of repeating characters within a string
// forbar becomes fobar, aaaa becomes a, etc...
void CConvVarZcp::CleanDoubles(unsigned char *dirty_string, unsigned char *clean_string)
{
    if(strlen(char *)dirty_string) == 0)
        return;

    unsigned char *clean_string_ptr = clean_string;
    *clean_string_ptr = *dirty_string;
    while(*dirty_string)
    {
        dirty_string++;
        if(*dirty_string != *clean_string_ptr)
        {
            clean_string_ptr++;
            *clean_string_ptr = *dirty_string;
        }
    }
    *clean_string_ptr = '\0';
    return;
}

// this is the actual function that translates from variant to encoded form
// via the m_group_array array which contains the group assignments for the
// IPA characters
// narrative paragraph number 3.1.6
// narrative paragraph number 3.2.2
void CConvVarZcp::ConvertVarZcp(unsigned char *variant, char *group)
{
    char *groupStringPtr = group;
    while (*variant)
    {
        *groupStringPtr = m_group_array[(unsigned char)*variant];
        variant++;
        groupStringPtr++;
    }
    *groupStringPtr = '\0';
    return;
}

// this function converts from variants to groups now called (encoded forms or codes)
// and writes out the encoded groups file
// narrative paragraph number 3.2.2
bool CConvVarZcp::Groupify(CString *source, CString *destination, CString *log)
{
    FILE *s, *d; // s is the source file, d is the destination file
    FILE *logfile; // this is the error log for weird characters in the groups
    //////////////////////////////////////
    // these are used for the progress bar

```

```

m_dest_group_source_box.GetWindowText(source);
m_dest_group_dest_box.GetWindowText(destination);

if(!source.IsEmpty())
{
    MessageBox("You need to specify a source file to be sorted");
    return;
}

if(!destination.IsEmpty())
{
    MessageBox("You need to specify a destination file to be sorted");
    return;
}

if(source == destination)
{
    MessageBox("File names must be different.");
    return;
}

else
{
    // this is the actual function that does the de-duplicating
    // the 1 enables this to work on groups
    status = DeDupVariants(source, destination, 1);
}

if(status == false)
    MessageBox("A problem occurred while deduping groups");
else
    MessageBox("Finished de-duplicating groups.");

return;

// this button handler calls the function that encodes the
// variants.
// narrative paragraph number 3.2.2
void CConvVarZcp::OnGroupFile()
{
    CString source, destination, log;
    bool status;

    // do progress bar stuff
    m_progress_bar.SetStep(1);
    m_progress_bar.SetRange(0,100);
    m_progress_bar.SetPos(0);

    // get the file names from the interface
    m_source_file_for_groups_box.GetWindowText(source);
    m_dest_file_for_groups_box.GetWindowText(destination);

    if(source == destination)
    {
        MessageBox("File names must be different.");
        return;
    }
    else
    {
        // this little bunch of code creates a log file based on the culture of the
        // file being processed. Actually it takes the filename and adds the _log
        // at the end.
        m_source_file_for_groups_box.GetWindowText(log);
        log = log.Append(".log");
    }
}

```

```

DWORD interval_ctr = 0;
DWORD prev_interval = 0;
DWORD file_size = 0;
DWORD interval;

// ////////////////////////////////////////
char offset_buffer[MAX_JMWG_SIZE + 1];
unsigned char variant_buffer[MAX_JMWG_SIZE + 1];
unsigned char clean_variant_buffer[MAX_JMWG_SIZE + 1];
char group_buffer[MAX_JMWG_SIZE + 1];

long int total_variants = 0;
long int total_groups = 0;
unsigned long int current_file_pos;
int i; // generic counter
int nan_errors(0); // error counter
bool status = true;

// clear out the buffers
for(i = 0; i < (MAX_JMWG_SIZE + 1); i++)
{
    variant_buffer[i] = NULL;
    clean_variant[i] = NULL;
    group_buffer[i] = NULL;
    offset_buffer[i] = NULL;
}

// open the files
if (fd = fopen(L"CTSTR"destination, "w") == NULL)
{
    MessageBox("Could not open the destination file for writing");
    return false;
}

if (!logfile = fopen(L"CTSTR"log, "w") == NULL)
{
    MessageBox("Could not open the error log file for writing");
    return false;
}

if (fd = fopen(L"CTSTR"source, "rb") == NULL)
{
    MessageBox("Could not open source file for reading");
    return false;
}
else
{
    // this stuff is to initialize the progress bar
    file_size = _filelength_filenis(i);
    if(file_size)
    {
        if(file_size > 100.00)
            interval = file_size / 100.00;
        else
            interval = 100.00 / file_size;
    }
    else
        interval = 0;
}

// ////////////////////////////////////////
// convert to groups and write the temporary group file
while(!feof(offset_buffer, 1, 0)) && (fgets((char *)variant_buffer, MAX_JMWG_SIZE, s))

```

CONVVA-1.CPP 3-24-98 12:17p

```

total_variants++;
current_file_pos = ftell(s);
interval_ctr = current_file_pos / interval;
if(interval_ctr > prev_interval)
{
    // progress bar steps(i);
    prev_interval = interval_ctr;
}

variant_buffer[strlen((char *)variant_buffer) - 2] = NULL;

// uncomment these two lines if you want to remove repeating letters
// dont forget to comment out the following line
CleanDoubles(variant_buffer, clean_variant);
ConvertVar2grp(clean_variant, group_buffer); // convert to a group string

// uncomment this line if you don't want to remove repeating letters
// dont forget to comment out the previous two lines
// ConvertVar2grp(variant_buffer, group_buffer); // convert to a group string

if(!isodd(group_buffer))
{
    // write to file here
    total_groups++;
    fputs(offset_buffer, d);
    fputs(group_buffer, d);
    fputs("\n", d);
}
else
{
    // write to the log file
    fputs((const char *)variant_buffer, logfile);
    fputs(" ", logfile);
    fputs(group_buffer, logfile);
    fputs("\n", logfile);
    nan_errors++;
}

// clear out the group buffer, this may not really be necessary
for(i = 0; i < (MAX_JMWG_SIZE + 1); i++)
    group_buffer[i] = NULL;

// ////////////////////////////////////////
// put stats in the UI
char tot_variants[20] = {0};
char tot_groups[20] = {0};
char tot_errors[20] = {0};
sprintf(tot_variants, "%d", total_variants);
sprintf(tot_groups, "%d", total_groups);
sprintf(tot_errors, "%d", nan_errors);
// note: despite their names, these boxes no longer contain the number of
// unique variants or groups, yet
// num unique variants box2 SetWindowText(tot_variants);
// num unique groups created box SetWindowText(tot_groups);
// num_errors_box SetWindowText(tot_errors);

// ////////////////////////////////////////
fclose(s);
fclose(d);

```

Page 5 of 10


```

// the prev buffers
}

// put the offset in the unique set
offset = stol(prev_offset_buff);
offsets.insert(offset);

while(types(curr_offset_buff,1,1,1)) && (types(variant_buff,MAX_VARS_SIZE,1))
{
    // progress bar stuff
    current_file_position = tell(a);
    interval_ctr = current_file_position / interval;
    if(interval_ctr > prev_interval)
    {
        m_progress_bar.StepIt();
        prev_interval = interval_ctr;
    }

    num_variants_read++; // increment the number of variants read in counter
    variant_buff[length - 1] = NULL; // get rid of the newline
    if(strlen(prev_variant_buff,variant_buff) == 0) // variants are the same
    {
        offsets.insert(stol(curr_offset_buff));
        num_offsets_accumulator += num_offsets;
    }
    else // variants are different
    {
        // write out the variant
        write_length = strlen(prev_variant_buff) + 1; // the + 1 is to account for the null we will add later
        variant_length_accumulator += strlen(prev_variant_buff);
        if (write_length - 1 > max_variant_length)
            max_variant_length = (write_length - 1);
        if (write_length - 1 < min_variant_length)
            min_variant_length = (write_length - 1);
        fwrite(write_length,sizeof(int),1,d); // write out the length of the variant
        fwrite(prev_variant_buff,d); // write out the actual variant
        fprintf("0",d); // null terminate the variant/group to make it easier to read in later
        num_unique_variants++; // increment the unique variants counter

        // write out the number of offsets to follow
        set_size = offsets.size();
        fwrite(set_size,sizeof(int),1,d);
        if(set_size > max_offsets)
            max_offsets = set_size;
        if(set_size < min_offsets)
            min_offsets = set_size;

        // write out the set of offsets
        for(offsets_iterator = offsets.begin(); offsets_iterator != offsets.end(); offsets_iterator++)
            fwrite(*offsets_iterator,sizeof(unsigned long int),1,d);

        // empty everything
        offsets.erase(offsets.begin(), offsets.end());

        // close the files and return
        fclose(a);
        fclose(d);

        char buffer[15] = {0};

        if(flag == 1)
        {
            // update the display for groups
            m_num_variants_read = buffer, 10;
            m_num_groups_read_in_box.SetWindowText(buffer);
            sprintf(buffer,"0",15);
            m_num_unique_variants.SetWindowText(buffer);
            m_unique_groups_box.SetWindowText(buffer, 10);
            sprintf(buffer,"0",15);
            m_grp_len_max_box.SetWindowText(buffer);
            m_grp_len_max_box.SetWindowText(buffer, 10);
            sprintf(buffer,"0",15);
            m_grp_len_min_box.SetWindowText(buffer);
            m_grp_len_min_box.SetWindowText(buffer, 10);
            sprintf(buffer,"0",15);
            m_grp_len_max_box.SetWindowText(buffer);
            m_grp_len_max_box.SetWindowText(buffer, 10);
            m_grp_max_offsets.SetWindowText(buffer);
            m_grp_max_offsets.SetWindowText(buffer, 10);
        }
    }
}

```

```

// the prev buffers
}

// put the offset in the unique set
offset = stol(prev_offset_buff);
offsets.insert(offset);

while(types(curr_offset_buff,1,1,1)) && (types(variant_buff,MAX_VARS_SIZE,1))
{
    // progress bar stuff
    current_file_position = tell(a);
    interval_ctr = current_file_position / interval;
    if(interval_ctr > prev_interval)
    {
        m_progress_bar.StepIt();
        prev_interval = interval_ctr;
    }

    num_variants_read++; // increment the number of variants read in counter
    variant_buff[length - 1] = NULL; // get rid of the newline
    if(strlen(prev_variant_buff,variant_buff) == 0) // variants are the same
    {
        offsets.insert(stol(curr_offset_buff));
        num_offsets_accumulator += num_offsets;
    }
    else // variants are different
    {
        // write out the variant
        write_length = strlen(prev_variant_buff) + 1; // the + 1 is to account for the null we will add later
        variant_length_accumulator += strlen(prev_variant_buff);
        if (write_length - 1 > max_variant_length)
            max_variant_length = (write_length - 1);
        if (write_length - 1 < min_variant_length)
            min_variant_length = (write_length - 1);
        fwrite(write_length,sizeof(int),1,d); // write out the length of the variant
        fwrite(prev_variant_buff,d); // write out the actual variant
        fprintf("0",d); // null terminate the variant/group to make it easier to read in later
        num_unique_variants++; // increment the unique variants counter

        // write out the number of offsets to follow
        set_size = offsets.size();
        fwrite(set_size,sizeof(int),1,d);
        if(set_size > max_offsets)
            max_offsets = set_size;
        if(set_size < min_offsets)
            min_offsets = set_size;

        // write out the list of offsets
        for(offsets_iterator = offsets.begin(); offsets_iterator != offsets.end(); offsets_iterator++)
        {
            size_t check = fwrite(*offsets_iterator,sizeof(unsigned long int),1,d);
            if(check != 1)
            {
                MessageBox("hold your horses!");
            }
        }

        // set the previous buffers to hold the new values
        offsets.erase(offsets.begin(), offsets.end()); // empty the list
        strcpy(prev_variant_buff, variant_buff); // set prev buffers
        strcpy(prev_offset_buff, curr_offset_buff); //
    }
}

```

```

status(buffer, '0', 15);
wchar_t* offset;
m_group_offsets.push_back(SetupIndex(buffer));
status(buffer, '0', 15);

sprintf(buffer, "%2d", (variant_length_accumulator / (double)num_unique_variants));
m_group_len_avg.push_back(SetupIndex(buffer));
status(buffer, '0', 15);
return true;
}

// this is the buffer handler that produces the final
// *.idx and *.vec files, these are index and maps into the
// database the *.nam files
// narrative paragraph number 3.2.3
// narrative paragraph number 3.2.4
void CONWVarZorp::GenerateIndexes()
{
    CString source_idxfile, offsetfile;
    bool status;

    //do progress bar stuff
    m_progress_bar.SetStep(1);
    m_progress_bar.SetRange(0, 100);
    m_progress_bar.SetPos(0);

    //get the file names from the interface
    m_gen_indexes_source_box.GetWindowText(source);
    m_gen_indexes_idx_box.GetWindowText(idxfile);
    m_gen_indexes_vec_box.GetWindowText(offsetfile);

    if(source.IsEmpty())
    {
        MessageBox("You must specify a source file for input.");
        return;
    }
    if(idxfile.IsEmpty())
    {
        MessageBox("You must specify an index file for output.");
        return;
    }
    if(offsetfile.IsEmpty())
    {
        MessageBox("You must specify an offsets file for output.");
        return;
    }
    if((source == idxfile) || (source == offsetfile) || (idxfile == offsetfile))
    {
        MessageBox("The files you specified should have different names.");
        return;
    }

    // this is the actual index file generation function
    status = GenerateIndexes(source, idxfile, offsetfile);
    if(status == false)
    {
        MessageBox("A problem occurred while generating the indexes.");
    }
    else
    {
        MessageBox("Finished generating the indexes.");
    }
    return;
}

// This function generates the final index files *.idx and *.vec
// narrative paragraph number 3.2.3

```

```

// narrative paragraph number 3.2.4
bool CONWVarZorp::GenerateIndexes(CString *source, CString *idxfile, CString *offsetfile)
{
    FILE *src, *idx, *ofst;
    DWORD current_file_position = 0;
    DWORD current_idx_pos = 0;
    DWORD prev_interval = 0;
    DWORD file_size = 0;
    DWORD interval;

    int group_len(0);
    char group[31] = {0};
    long unsigned int offset_file_position(0);
    int nam_offsets(0);
    unsigned int offset(0);

    int i(0); // generic counter used in for loops

    // open the files for I/O
    if ((idx = fopen(LPCSTR) == NULL) || (offsetfile = fopen(LPCSTR) == NULL))
    {
        MessageBox("Could not open the index file for writing");
        return false;
    }
    if ((ofst = fopen(LPCSTR) == NULL) || (idxfile = fopen(LPCSTR) == NULL))
    {
        MessageBox("Could not open the offsets file for writing");
        return false;
    }
    if ((src = fopen(LPCSTR) == NULL) || (idxfile = fopen(LPCSTR) == NULL))
    {
        MessageBox("Could not open source file for reading");
        return false;
    }
    else
    {
        // this stuff is for the progress bar
        file_size = _filenametlength(idxfile);
        if(file_size)
        {
            if(file_size > 100)
            {
                interval = file_size / 100.00;
            }
            else
            {
                interval = 100 / file_size;
            }
        }
        else
        {
            interval = 0;
        }
        while(!feof(src))
        {
            // more progress bar stuff
            current_file_position = ftell(src);
            interval_ctr = current_file_position / interval;
            if(interval_ctr > prev_interval)
            {
                m_progress_bar.Step();
                prev_interval = interval_ctr;
            }
        }
        group_len = (getc(src)); // read the group length
        fgets(group, group_len + 1, src); // read the group
    }
}

```

```

bool status;
CString source, destination, log;
CString logfile, offsetfile;

m_auto_flag = true; // set this so that some of the AddressBoxes do not display
////////////////////
// Generate groups step
////////////////////

// do progress bar stuff;
m_progress_bar.SetStep(1);
m_progress_bar.SetRange(0,100);
m_progress_bar.SetPos(0);

// get the file names from the interface
m_source_file_for_groups_box.GetWindowText(source);
m_dest_file_for_groups_box.GetWindowText(destination);

if(source == destination)
{
    AfxMessageBox("File names must be different. \nthis step was not performed successfully.");
    m_OK = false;
    return;
}
else
{
    m_source_file_for_groups_box.GetWindowText(log);
    log = log.SpanBefore("\\");
    log += "_log.txt";

    status = Groupify(source, destination, log);
    UpdateWindow();
}

if(status == false)
{
    AfxMessageBox("A problem occurred while converting to groups. \nthe rest of the steps were not performed.");
    return;
}
////////////////////
// Sort the groups step
OnSortGroupsBtn();
UpdateWindow();
////////////////////
// De-duplicate groups step
////////////////////

// do progress bar stuff
m_progress_bar.SetStep(1);
m_progress_bar.SetRange(0,100);
m_progress_bar.SetPos(0);

// get file names from the UI
m_dedup_group_source_box.GetWindowText(source);
m_dedup_group_dest_box.GetWindowText(destination);

if(source.IsEmpty()) // shouldn't happen but its here anyway
{
    AfxMessageBox("You need to specify a source file to be sorted. \nthis step was not performed successfully.");
    return;
}

```

```

fpact(group_idx); // write the group to the index file
for(i = strlen(group); i < MAX_NAME_SIZE + 1; i++) // write padding
    fpact('0', idx);

freopen(offsets, "a", fopen(tmpl, "a+")); // read the number of offsets to follow
offset_file_position = ftell(offset); // get start address of offsets file here
fwrite(offset_file_position, sizeof(unsigned int), 1, idx); // write start address of offsets file
-- 0 index file
fwrite(num_offsets, sizeof(unsigned int), 1, idx); // write the number of offsets to follow

for(i = 0; i < num_offsets; i++) // loop for as many offsets as there are
{
    fread(offsets, sizeof(unsigned int), 1, src); // read an offset
    fwrite(offset, sizeof(unsigned int), 1, dst); // write the offsets to the offsets file
}

fclose(src);
fclose(idx);
fclose(dst);
return true;
}

// this function is used to get rid of the intermediate files that
// are produced during the database building phase.
// these files are the *.grp *.var files
bool ConvertCorp::DeleteIntermediateFiles()
{
    log handle;
    int status(0);
    _finddata_t fileinfo;
    char filespec[6] = "*.var"; // variant files

    //delete the *.var files
    handle = _findfirst(filespec, &fileinfo);
    if(handle != -1)
    {
        status = remove(fileinfo.name);
        while(_findnext(handle, &fileinfo) != -1)
        {
            status = remove(fileinfo.name);
        }
    }

    // remove the *.grp files
    sprintf(filespec, "*.grp");
    handle = _findfirst(filespec, &fileinfo);
    if(handle != -1)
    {
        status = remove(fileinfo.name);
        while(_findnext(handle, &fileinfo) != -1)
        {
            status = remove(fileinfo.name);
        }
    }

    return status == 0; // this works around the old bool problem
}

// this function is used to automate the entire process for one culture.
// it is supposed to be called when all the filenames have been specified
void ConvertCorp::DoAllSteps()
{

```



```

    IDC_Control (pdx, IDC_PROCESSING_LOG, mc_processing_log);
    IDC_Control (pdx, IDC_HISPANIC_VARIANTS, mc_hispanic_variants);
    IDC_Control (pdx, IDC_HISPANIC_NAMES, mc_hispanic_names);
    IDC_Control (pdx, IDC_CHINESE_NAMES, mc_chinese_names);
    IDC_Control (pdx, IDC_CHINESE_VARIANTS, mc_chinese_variants);
    IDC_Control (pdx, IDC_ARABIC_NAMES, mc_arabic_names);
    IDC_Control (pdx, IDC_ARABIC_VARIANTS, mc_arabic_variants);
    IDC_Control (pdx, IDC_ANGLO_NAMES, mc_anglo_names);
    IDC_Control (pdx, IDC_ANGLO_VARIANTS, mc_anglo_variants);
    IDC_Control (pdx, IDC_ARABIC_RULES, mc_arabic_rules);
    IDC_Control (pdx, IDC_CHINESE_RULES, mc_chinese_rules);
    IDC_Control (pdx, IDC_HISPANIC_RULES, mc_hispanic_rules);
    IDC_Control (pdx, IDC_BROWSE, mc_browse);
    IDC_Control (pdx, IDC_LOG, m_log);
    ///AFX_DATA_END

    //Initial Message Map (OnCommandMsg, CmdMsg)
    ///AFX_MSG_MAP (OnCommandMsg)
    ON_WM_SYSCOMMAND()
    ON_WM_PAINT()
    ON_WM_QUERYCaret()
    ON_BT_CLICKED(IDC_BROWSE, OnBrowse)
    ON_BT_SELECTED(IDC_ARABIC_RULES, OnGetFocusArabicRules)
    ON_BT_SELECTED(IDC_ANGLO_RULES, OnGetFocusAngloRules)
    ON_BT_SELECTED(IDC_CHINESE_NAMES, OnGetFocusChineseNames)
    ON_BT_SELECTED(IDC_HISPANIC_NAMES, OnGetFocusHispaniNames)
    ON_BT_SELECTED(IDC_ANGLO_NAMES, OnGetFocusAngloNames)
    ON_BT_SELECTED(IDC_ARABIC_VARIANTS, OnGetFocusArabicVariants)
    ON_BT_SELECTED(IDC_ARABIC_NAMES, OnGetFocusArabicNames)
    ON_BT_SELECTED(IDC_CHINESE_VARIANTS, OnGetFocusChineseVariants)
    ON_BT_SELECTED(IDC_CHINESE_NAMES, OnGetFocusChineseNames)
    ON_BT_SELECTED(IDC_HISPANIC_VARIANTS, OnGetFocusHispaniVariants)
    ON_BT_SELECTED(IDC_HISPANIC_NAMES, OnGetFocusHispaniNames)
    ON_BT_SELECTED(IDC_PROCESSING_LOG, OnGetFocusProcessingLog)
    ON_BT_SELECTED(IDC_Simplified_RULES, OnGetFocusSimplifiedRules)
    ON_WM_DESTROY (IDC_EVERYTHING, OnEverything)
    ///AFX_MSG_MAP_END

    DO_MESSAGE_MAP()

    // OnCommandMsg message handlers
    void OnCommandMsg (CmdMsg)
    {
        CmdMsg: OnInitialDialog();

        // Add "About..." menu item to system menu.

        // IDM_ABOUTBOX must be in the system command range.
        ASSERT((IDM_ABOUTBOX & 0xFFF0) == IDM_ABOUTBOX);
        ASSERT(IDM_ABOUTBOX < 0xF000);

        CMenu* pSystemMenu = GetSystemMenu(FALSE);
        if (pSystemMenu != NULL)
        {
            CString strAboutMenu;
            strAboutMenu.LoadString(IDS_ABOUTBOX);
            if (!strAboutMenu.IsEmpty())
            {
                pSystemMenu->AppendMenu(MF_SEPARATOR);
                pSystemMenu->AppendMenu(MF_STRING, IDM_ABOUTBOX, strAboutMenu);
            }
        }
    }
}

```


Page 4 of 12

NAME=1 CPP 3-24-98 12:17p

```

int error, // something used for error processing
int name, // number of variants generated by the ProcessVariants() function
int maxAngloV = 0; // greatest number of variants generated by one anglo name
int maxArabicV = 0; // greatest number of variants generated by one arabic name
int maxChineseV = 0; // greatest number of variants generated by one chinese name
int maxHispanicV = 0; // greatest number of variants generated by one hispanic name

unsigned long arabic = 0; // number of names classified as arabic
unsigned long chinese = 0; // number of names classified as chinese
unsigned long hispanic = 0; // number of names classified as hispanic

unsigned long angloOffset = 0, arabicOffset = 0, chineseOffset = 0, hispanicOffset = 0; // offsets into the
//> name files
unsigned long total = 0; // total number of names read in from the input file, some of these can be invalid
unsigned long valid = 0; // total number of valid names read in from the input file

//> aux code
// these four variables will keep track of the total number of variants
unsigned long total_anglo_variants(0);
unsigned long total_arab_variants(0);
unsigned long total_chinese_variants(0);
unsigned long total_hispanic_variants(0);

// these are used to keep track of which name produced the most variants :-))
CString most_prolific_anglo;
CString most_prolific_arab;
CString most_prolific_chinese;
CString most_prolific_hispanic;

// an enumerated type, that records the result of
// the classification done by nas
enum nam_language lang;

CString rulesetname; // used to pass the name of the rule set to the processVariants() function so that
// we can tell which rule set was being used when the regexp error occurred.
// this was the easiest way to do it.

// =====
// this stuff is for the progress bar
DWORD interval_ctr = 0;
DWORD prev_interval = 0;
DWORD file_size = 0;
DWORD interval;
unsigned long int current_file_pos;

// =====
// Sort the file

msg.Format("Sorting the names file!");
m_log.AddString(msg);
UpdateProgressBar();

SortNames();

msg.Format("Finished sorting the names file!");
m_log.AddString(msg);
UpdateProgressBar();

// =====
msg.Format("Processing file is...", m_input_names );
m_log.AddString(msg);

```

Page 5 of 12

```

nameVariants = variants->size();
nameSetA = ruleSet->GetSimplifiedCodeArrayForString( vinput, stdCodeArray, MAX_LEN );
// write out the record to the names file
fwrite( &id, sizeof( unsigned long ), 1, fout );
fwrite( groupID, sizeof( unsigned char ), 7, fout );
fwrite( input, fout );
fwrite( '\0', fout );
offsetAddendum = sizeof( unsigned long ) + strlen( input ) + 8; // length of id, groupID, name, and on
//, e zero

fwrite( usingCodeArray, sizeof( unsigned char ), numSetA + 1, fout );
offsetAddendum += numSetA + 1;

fwrite( GetVowelStatus( variants ), fout );
offsetAddendum++;
for( i = 0; i < numVariants; i++ )
{
    k = 0;
    len = ( ( *variants )[i] ).GetLength();
    if( len > 0 )
    {
        // skip over leading vowels
        while( k < len && IS_VOWEL( ( *variants )[i][k] ) ) k++;

        // NOTE: the next two lines must be commented in or out together, as
        // the length must be recalculated iff the Strip routine is called:
        StripAndWrite( &( *variants )[i] );
        //len = ( ( *variants )[i] ).GetLength();

        // output the leading consonants to the names file
        if( k < len && ( ( *variants )[i][k] ) != '\0' ) {
            k++;
            len--;
            fwrite( &( *variants )[i][k], 1, len, fout );
            offsetAddendum++;
        }

        // output the variant and offset to the variants file
        sprintf( vout, "%10.10s\t", offset, ( *variants )[i] );
    }
    else
    {
        msg.Format( "ERROR: NULL variants generated for '%s', input: ",
            m_log.AddString( msg );
        sprintf( processing, "%s\t", msg );
    }
}

fwrite( '\0', fout );
offsetAddendum++;
// update the offset based on the total size of the record just written
*offset += offsetAddendum;
delete variants;
return numVariants;
}

// narrative paragraph number 3.1
// This method goes through the name input file line by line, validates each name,
// classifies it, and processes the name and its variants into the names and variants files.
// Statistics are gathered and reported regarding variant numbers, classification,
// and local input and valid names
void CNameSetDlg::ProcessFile()
{
    char input[MAX_IN_LEN], vinput[MAX_IN_LEN], groupID[7]; // input buffers for the input names and ORG group
    // id strings
    CString msg; // generic buffer used for writing out messages to the list boxes or log files
}

```

MKNAME~1.CPP 3-24-98 12:17p

```

UpdateWindow();
printf( processing, "%s\n", msg );

// closes the old input file and points to the new sorted
// input file for processing.
fclose(inputNames);
inputNames = NULL;

mc_input_names GetWindowNextIn( input_names );
BOOL ok = OpenFile(inputNames, "input_names", "r");
if(!ok)
{
    ReportError( INPUT_OPEN_ERROR, (LPCWSTR)mc_input_names, 0 );
    return;
}

// the inputNames file is opened with the InitialisesFiles() function before the
// ProcessFile() function is called. this is done in the OnOK() function.
// at this point all the files should be open.

////////////////////////////////////
// progress bar stuff
// since I set the range of the progress bar to be 100 units long.
file_size = _filenlength(_fileno(inputNames));
if(file_size)
{
    if(file_size > 100.00)
        interval = file_size / 100.00;
    else
        interval = 100.00 / file_size;
}
else
    interval = 0;
//
////////////////////////////////////

// get the names
while( GetStd( input, MAX_IN_LEN, inputNames ) != NULL )
{
    total++; // this counts the total number of names read in, the names can be invalid
    // so total may not necessarily equal the number of valid records read in

    //////////////////////////////////////
    // progress bar stuff
    // this is what controls the advance of the progress bar
    current_file_pos = _ftell(inputNames);
    interval_ctr = current_file_pos / interval;
    if(interval_ctr > prev_interval)
    {
        m_db_progress_bar.StepIt();
        prev_interval = interval_ctr;
    }
    //
    // make sure there is a space between the groupID and the name
    if(input[6] != ' ')
    {
        ReportError(6, input, 0);
    }
    else
    {
        strcpy( groupID, input, 6 ); // the group id is just a 6 digit id that QED uses to group names
        groupID[6] = 0;
        if( ! error = CleanGroupID( groupID ) ) != NO_ERROR )
        {

```

```

        ReportError( error, groupID, total+1 );
    }
    else
    {
        // input? is where the name starts
        tch_strcpy( input? );
        string( input? );
        if( ! error = CleanName( input? ) ) != NO_ERROR )
        {
            ReportError( error, input?, total+1 );
        }
        else
        {
            //this line determines what culture a name is classified as
            lang = mylang_analyze( input? );
            vinput[0] = '\0'; // name must be space delimited for variant generation
            string( vinput, input? );
            string( vinput, input? );

            // here is where the variants are generated
            // numv is the number of variants produced
            // this line processes for the output rule set, this is done to all records by default

            rulesetname = "anglo rules";
            numv = ProcessVariants( input?, vinput, groupID, valid, anglobuleSet,
            langidoffset, angloOut, angloOut, rulesetname, total + 1 );

            total_anglo_variants += numv;

            if( numv > maxanglov )
            {
                maxanglov = numv;
                most_prolific_anglo = vinput;
            }

            // if a name is classified as arabic this stuff is done
            // Chinese and Hispanic counterparts follow as options
            if( lang == NAS_ARABIC )
            {
                rulesetname = "arabic rules";
                numv = ProcessVariants( input?, vinput, groupID, valid, arabibuleSet,
                arabicoffset, arabicOut, arabicOut, rulesetname, total + 1 );

                total_arab_variants += numv;

                if( numv > maxarabicv )
                {
                    maxarabicv = numv;
                    most_prolific_arab = vinput;
                }
            }
            else if( lang == NAS_CHINESE )
            {
                rulesetname = "chinese rules";
                numv = ProcessVariants( input?, vinput, groupID, valid, chinesebuleSet,
                chineoffset, chineseOut, chineseOut, rulesetname, total + 1 );

                total_chinese_variants += numv;

                if( numv > maxchinesev )
                {

```

```

machineseV = name;
most_prolific_chinese = vinput;
}

// Chinese++
else if ( lang == MAS_HISPANIC )
{
    rulesetname = "hispanic.rules";
    name = ProcessVariants( input, vinput, groupID, valid, hispanicruleset,
        hispanicoffset, hispanicout, hispanicout, rulesetname, total + 1 );
    total_hispanic_variants += name;

    if ( name == most_prolific_hispanicV )
    {
        most_prolific_hispanic = name;
        most_prolific_hispanic = vinput;
    }

    hispanic++;
}

valid++; // this counts the number of valid records
}
}

// print out statistics
msg.Format( "Done. Retrieved %d valid records out of %d records processed", valid, total );
m_log.AddString( msg );
fprintf( processing, "%s\n", msg );

msg.Format( "Id names were classified as Arabic.", nArabic );
m_log.AddString( msg );
fprintf( processing, "%s\n", msg );

msg.Format( "Id names were classified as Chinese.", nChinese );
m_log.AddString( msg );
fprintf( processing, "%s\n", msg );

msg.Format( "Id names were classified as Hispanic.", nHispanic );
m_log.AddString( msg );
fprintf( processing, "%s\n", msg );

msg.Format( "Id was the maximum number of Anglo variants produced by %s", most_prolific_anglo );
m_log.AddString( msg );
fprintf( processing, "%s\n", msg );

msg.Format( "Id was the maximum number of Arabic variants produced by %s", most_prolific_arabi );
m_log.AddString( msg );
fprintf( processing, "%s\n", msg );

msg.Format( "Id was the maximum number of Chinese variants produced by %s", most_prolific_chine );
m_log.AddString( msg );
fprintf( processing, "%s\n", msg );

msg.Format( "Id was the maximum number of Hispanic variants produced by %s", most_prolific_his );
m_log.AddString( msg );
fprintf( processing, "%s\n", msg );

msg.Format( "Id Anglo variants were produced", total_anglo_variants );
m_log.AddString( msg );

```

```

fprintf( processing, "%s\n", msg );
msg.Format( "Id Arab variants were produced", total_arab_variants );
m_log.AddString( msg );
fprintf( processing, "%s\n", msg );

msg.Format( "Id Chinese variants were produced", total_chinese_variants );
m_log.AddString( msg );
fprintf( processing, "%s\n", msg );

msg.Format( "Id Hispanic variants were produced", total_hispanic_variants );
m_log.AddString( msg );
fprintf( processing, "%s\n", msg );

m_log.SetTopIndex( m_log.GetCount() - 1 ); // scroll down to the end of the list

// close all files
fclose( inputnames );
fclose( processing );
fclose( angloout );
fclose( arabicout );
fclose( arabicout );
fclose( chineseout );
fclose( chineseout );
fclose( hispanicout );
fclose( hispanicout );

////////////////////
// get rid of the sorted input file here
long handle;
int status(0);
_fwrite_t fileinfo;

handle = _findfirst( LACTSRM_input_names, &fileinfo );
if (handle != -1)
{
    status = remove(fileinfo.name);
}

// at this point the sorted file is deleted
////////////////////

// get rid of dangling pointers
inputnames = NULL;
processing = NULL;
angloout = NULL;
arabicout = NULL;
arabicout = NULL;
chineseout = NULL;
chineseout = NULL;
hispanicout = NULL;
hispanicout = NULL;

// This method initializes a ruleset pointed to by **ruleset, which is contained
// in the file frame. fp points to the previously opened simplified rules file.
// This is added to the rule set if it is initialized properly.
// Any errors are reported.
BOOL OpenRules( long: InitializeRuleset( Ruleset **ruleset, CString frame, FILE *fp )
{
    CString temp_str;
    BOOL result = true;
}

```

```

if ( fname.IsEmpty() == FALSE )
{
    rewind( fp );
    *ruleset = new Ruleset( fname );
    temp_str.Format( "Compiling Rules File %s", fname );
    m_log.AddString( temp_str );
    if ( (*ruleset)-header() )
    {
        if ( (*ruleset)-addSimplifiedRules( fp, processing, NLL ) == FALSE )
        {
            result = false;
            temp_str.Format( "ERROR: Error adding simplified rules to rules file %s", fname );
            m_log.AddString( temp_str );
        }
    }
    else
    {
        result = false;
        temp_str.Format( "ERROR: Missing Rules File %s", fname );
        m_log.AddString( temp_str );
        if ( processing ) printf( processing, "%s\n", temp_str );
    }
}
else
{
    temp_str.Format( "ERROR: Missing rules file specification." );
    m_log.AddString( temp_str );
    result = false;
}
return result;
}

// This method initializes all the rules files, including the simplified rules file
bool CUnicodeLog::InitializeRules()
{
    CSString msg;
    bool result = true;
    FILE *fp;

    m_english_rules.Truncate();
    m_english_rules.Truncate();
    m_arabic_rules.Truncate();
    m_arabic_rules.Truncate();
    m_chinese_rules.Truncate();
    m_chinese_rules.Truncate();
    m_chinese_rules.Truncate();
    m_chinese_rules.Truncate();
    m_simplified_rules.Truncate();
    m_simplified_rules.Truncate();

    if ( m_simplified_rules.IsEmpty() )
    {
        fp = fopen( m_simplified_rules, "r" );
        if ( fp )
        {
            result = false;
            msg.Format( "ERROR: unable to open simplified rules file %s",
                m_simplified_rules.IsEmpty() ? "null" : m_simplified_rules );
            m_log.AddString( msg );
            if ( processing ) printf( processing, "%s\n", msg );
        }
        else
        {
            if ( result = InitializeRules( m_english_rules, m_english_rules, fp ) )
            {
                result = InitializeRules( m_arabic_rules, m_arabic_rules, fp );
                result = InitializeRules( m_chinese_rules, m_chinese_rules, fp );
            }
        }
    }
}

```

FILENAME1.CPP 3-24-98 12:17p

```

( result = InitializeRules( m_hispanic_rules, m_hispanic_rules, fp ) );
fclose( fp );
return result;
}

// This method initializes the entire process based on user hitting the button.
// All file names are grabbed from the interface.
// The progress bar is set up.
// I/O files are initialized.
// Rule sets are initialized.
// The names input file is processed.
void CUnicodeLog::CLICK()
{
    // TODO: Add extra validation here
    m_processing_log.GetWindowText( m_processing_log );
    m_hispanic_rules.GetWindowText( m_hispanic_rules );
    m_hispanic_rules.GetWindowText( m_hispanic_rules );
    m_chinese_rules.GetWindowText( m_chinese_rules );
    m_chinese_rules.GetWindowText( m_chinese_rules );
    m_arabic_rules.GetWindowText( m_arabic_rules );
    m_arabic_rules.GetWindowText( m_arabic_rules );
    m_english_rules.GetWindowText( m_english_rules );
    m_english_rules.GetWindowText( m_english_rules );
    m_simplified_rules.GetWindowText( m_simplified_rules );
    m_simplified_rules.GetWindowText( m_simplified_rules );

    // progress bar stuff
    m_db_progress_bar.SetStep(1);
    m_db_progress_bar.SetRange(0, 100);
    m_db_progress_bar.SetPos(0);

    m_log.ResetContent();
    if ( InitializeFiles() )
    {
        BeginWaitCursor();
        ProcessFile();
        EndWaitCursor();
    }
    //Dialog::CLICK();

    // A file browse request has been made, which invokes this method.
    // An active file CObject object and its type have been previously set.
    // And the dialog configures itself, and affects the right objects, accordingly.
    void CUnicodeLog::OnBrowse()
    {
        static char *filetypes[] = {
            "Rule Files (*.rul)|*.rul|All Files (*.*)|*.*|",
            "Doc Files (*.doc)|*.doc|None Files (*.nam)|*.nam|All Files (*.*)|*.*|",
            "None Files (*.nam)|*.nam|All Files (*.*)|*.*|",
            "Variant Files (*.var)|*.var|All Files (*.*)|*.*|",
            "Log Files (*.log)|*.log|Text Files (*.txt)|*.txt|All Files (*.*)|*.*|", 0 };

        // TODO: Add your control notification handler code here
        if ( activeFile )
        {

```

```

CFileDialog file_open_dialog(true,
    "txt",
    NULL,
    OFN_READONLY | OFN_HIDEPARENT,
    file_type(activefile_type),
    NULL);

if (file_open_dialog.IsModal() == IDOK)
{
    activefile->SetWindowText(file_open_dialog.GetPathName());
    m_log.RefreshContent();
}
else m_log.AddString("No active file component");
}

// this function is no longer necessary since sorting is now automatically done
// in the process stage.
void CMainMenuDlg::OnSort()
{
    // TODO: Add your control notification handler code here
    m_input_names.GetWindowText(m_input_names);
    if (m_input_names.IsEmpty())
    {
        CSortDialog *sortD = new CSortDialog(this, m_input_names);
        sortD->ShowModal();
    }
    else MessageBox("No input file has been specified. Cannot sort.");
}

// as disabling this function for now
// the button that invokes this has been disabled
// and rendered invisible in the GUI
// it will be reactivated when the function works
void CMainMenuDlg::OnEverything()
{
    //MessageBox("This button has not been implemented yet.");
    // call the sort here
    //look(); // this is the function that starts the processing stuff

    // then we programmatically switch tabs to access the rest of the
    // the process stuff
    CTabCtrl *pTabCtrl = (CTabCtrl *)GetParent();
    //CConVar20rp *var20rp = pTabCtrl->GetPage(0);
    parent->GetPage(0)->ActivatePage(0, false);
    parent->GetPage(1)->ActivatePage(1, true);
    //parent->GetPage(1)->SetActivePage(1);
    //parent->GetPage(1)->SetActivePage(1);
}

// either switch to the other dialog box and do the processing
// or figure out some other way
}

bool CMainMenuDlg::SortNames()
{
    CString sortname, msg;
    int status = 0;
    FILE *sorted;

    char short_sortname[256] = {0};

```

MKNAME1.CPP 3-24-98 12:17p

```

char short_sortname[256] = {0};

m_input_names.GetWindowText(m_input_names);
if (m_input_names.IsEmpty())
{
    return false;
}
else
{
    // make file name for the sorted name file
    sortname = m_input_names.SpanExcluding(".");
    sortname += "sorted.txt";

    // long name short name stuff
    if (sorted = fopen(LPTSTR sortname, "w")) == NULL
    {
        MessageBox("Could not open a blank sorted names file");
        return false;
    }
    fclose(sorted);

    GetShortPathName(m_input_names, short_name, double(256));
    GetShortPathName(sortname, short_sortname, double(256));
    //
    //
    msg.Format("Sorting %s...", m_input_names);
    m_log.AddString(msg);
    BeginWaitCursor();

    status = _spawnl_P_WAIT,
    "sort.com",
    "/Q",
    short_name,
    "/B:10",
    short_sortname,
    NULL);

    EndWaitCursor();

    if (status != 0) // this is for spawn
    {
        msg.Format("Error sorting names file '%s' (%d)", m_input_names, status);
        m_log.AddString(msg);
        return false;
    }
    else
    {
        msg.Format("File sorted successfully. Saved as '%s'", sortname);
        m_log.AddString(msg);
        m_log.AddString("The names input file has been changed accordingly.");
        m_input_names.SetWindowText(sortname);
        UpdateWindow();
    }
    return true;
}

// The following OnSetFocus methods are invoked whenever a user clicks into
// one of the CEdit controls in the interface that represents the specification
// of a file name. activefile and activefiletype are set so that when the

```

Page 9 of 12


```

"report.com",
"/Q",
short_nameDB,
"/r:30",
short_surname,
NULL);

BreakCursor();

if(status != 0) // this is for spawl
//if( status == 0 ) // this is for createprocess
{
    msg.Format( "Error sorting names file '%s' (%d)", nameDB, error );
    parent->m_log.AddMsg( msg );
}
else
{
    msg.Format( "File sorted successfully. Saved as '%s'", surname );
    parent->m_log.AddMsg( msg );
    parent->m_log.AddMsg( "The names input file has been changed accordingly." );
    parent->m_input_names.SetDlgItemText( surname );
    EndDialog( 0 );
}
}
}
//
/*
bool CSortDialog::OnInitDialog()
{
    CDialog::OnInitDialog();
    // TODO: Add extra initialization here
    m_inputSpec.SetDlgItemText( nameDB );
    m_sorted_names.SetFocus();
    return FALSE; // return TRUE unless you set the focus to a control
    // EXCEPTION: OCX Property Pages should return FALSE
}
*/

```

```
// Copyright (C) 1998, Language Analysis Systems Inc.  
//  
// sdata.cip : source file that includes just the standard includes  
// "tdatail.pl.h" will be the pre-compiled header  
// sdata.obj will contain the pre-compiled type information  
//  
#include "sdata.h"
```

```

TabClientItem mask = TCIT_IDENT;
    GetStringException();
    Page_GetWindowPos(GetException());
    TabClientItem perfect = (char*)LACSTRN.GetString();
    InsertItem(Index, kTabClientItem);

    // Make the window conform to the tab control's client area.
    if (m_rect.IntersectTop(0))
    {
        // Get the tab area.
        CRect rectTab;
        GetClientRect(0, rectTab);

        // Get the client area.
        CRect clientRectTab;
        GetClientRect(m_rect);

        // Adjust the client area for the tab.
        m_rect.top += rectTab.bottom - rectTab.top;
        m_rect.bottom += rectTab.bottom - rectTab.top;
        m_rect.left = rectTab.left;
        m_rect.right = rectTab.left;

        // Adjust the client area for the tab control's placement.
        GetWindowPos(rectTab);
        GetParent(1)->ScreenToClient(rectTab);
        m_rect.top += rectTab.top;
        m_rect.bottom += rectTab.top;
        m_rect.left = rectTab.left;
        m_rect.right = rectTab.left;

    }
}

void CTabCtrlSheet::RemovePage(int nPage)
{
    ASSERT(nPage >= 0);
    ASSERT(nPage < GetPageCount());
    ASSERT(GetPageCount() > 1);

    RemovePage(GetPage(nPage));
}

void CTabCtrlSheet::RemovePage(CPropertyPage* pPage)
{
    ASSERT(pPage != NULL);
    ASSERT_KINDOF(CPropertyPage, pPage);
    ASSERT(GetPageCount() > 1);

    // See if this is the active page.
    int nIndex = GetActiveIndex(pPage);
    if (nIndex == GetActiveIndex())
    {
        // Deactivate the page.
        ActivatePage(pPage, FALSE);

        // Remove the page.
        DeleteItem(nIndex);
        m_Page.RemoveAt(nIndex);

        // See if this was the last page.
        if (nIndex == GetPageCount())
        {
            // If so, move to previous page for activation.
            nIndex--;
            ASSERT(nIndex >= 0);
        }
    }
}

```

```

ASSERT(pPage != NULL);
ASSERT_KINDOF(CPropertyPage, pPage);

// Scan array for matching pointer.
for (int nIndex = 0; nIndex < GetPageCount(); nIndex++)
{
    if (GetPage(nIndex) == pPage)
        return nIndex;
}

return -1; // Page not found
}

//-----
RESULT CTabCrSheet::QuerySiblings(WPARAM wParam, LPARAM lParam)
{
    LPARAM lResult = 0; // Default to no response

    for (int nIndex = 0; nIndex < GetPageCount(); nIndex++)
    {
        // See if this is the active page.
        if (GetPage(nIndex) != GetActivePage())
        {
            // If not, send the page a PSM_QUERYSIBLINGS message.
            RESULT lResult = GetPage(nIndex)->SendMessage(PSM_QUERYSIBLINGS, wParam, lParam);
            if (lResult != 0)
                // Stop when a page responds.
                break;
        }
    }

    return lResult;
}

//-----
bool CTabCrSheet::OnApply() const
{
    // See if current page will deactivate.
    if (!GetActivePage()->OnDeactivate())
        // Page refused to deactivate.
        return FALSE;

    // Call each page's OnApply to get results.
    for (int nIndex = 0; nIndex < GetPageCount(); nIndex++)
        if (!GetPage(nIndex)->OnApply())
            // Page refused the OnApply.
            return FALSE;

    return TRUE;
}

//-----
bool CTabCrSheet::OnQueryCancel() const
{
    // See if the current page allows the cancel.
    if (!GetActivePage()->OnQueryCancel())
        // Page refused to cancel.
        return FALSE;

    // Call each page's OnCancel handler.
    for (int nIndex = 0; nIndex < GetPageCount(); nIndex++)
        GetPage(nIndex)->OnCancel();

    return TRUE;
}
}
//-----

```

```

}

// Activate the appropriate page.
pPage = GetPage(nIndex);
ActivatePage(pPage, TRUE);
GetTabControl()->SetCursor(nIndex);
}
else
{
    // Remove the page.
    OnTestDestroy();
    m_Pages.RemoveAt(nIndex);
}

//-----
bool CTabCrSheet::SetActivePage(int nPage)
{
    ASSERT(nPage >= 0);
    ASSERT(nPage < GetPageCount());
    return SetActivePage(GetPage(nPage));
}

//-----
bool CTabCrSheet::SetActivePage(CPropertyPage* pPage)
{
    ASSERT(pPage != NULL);
    ASSERT_KINDOF(CPropertyPage, pPage);

    // Attempt to activate the page.
    if (pPage->SetActive())
    {
        // Page activated.
        ActivatePage(pPage, TRUE);
        GetTabControl()->SetCursor(GetPageIndex(pPage));
        return TRUE;
    }

    // Page refused to activate.
    return FALSE;
}

//-----
CPropertyPage* CTabCrSheet::GetPage(int nPage) const
{
    ASSERT(nPage >= 0);
    ASSERT(nPage < GetPageCount());
    return (CPropertyPage*)m_Pages.GetAt(nPage);
}

CPropertyPage* CTabCrSheet::GetActivePage() const
{
    int nPage = GetActiveIndex();
    if (nPage >= 1)
        return NULL;
    else
        return (CPropertyPage*)m_Pages.GetAt(nPage);
}

//-----
int CTabCrSheet::GetPageIndex(CPropertyPage* pPage) const
{

```

```

void CTabCrSheet::ActivatePage(int nPage, BOOL bActivate/* = TRUE */)
{
    ASSERT(nPage >= 0);
    ASSERT(nPage < GetPageCount());
    ActivatePage(GetPage(nPage));
}

void CTabCrSheet::ActivatePage(CPropertyPage* pPage, BOOL bActivate/* = TRUE */)
{
    ASSERT(pPage != NULL);
    ASSERT_VALID(pPage);
    // See if window is being activated or deactivated.
    if (bActivate)
    {
        // Activated by page.
        pPage->SetWindowPos(NULL, m_rect.TopLeft(), m_rect.Width(), m_rect.Height(), 0);
        // 2. Notify the page.
        mPage.mPage = (int) 0, PSI_SELECTIVE;
        pPage->SendMessage(WM_NOTIFY, WPARAM)mPage.mPageIdFrom, (LPARAM)mPageIdFrom;
        // 3. Show the window.
        pPage->ShowWindow(SW_SHOW);
    }
    else
    {
        // Deactivate the page.
        // 1. Notify the page.
        mPage.mPage = (int) 0, PSI_KILLACTIVE;
        pPage->SendMessage(WM_NOTIFY, WPARAM)mPage.mPageIdFrom, (LPARAM)mPageIdFrom;
        // 2. Position the window.
        pPage->SetWindowPos(HWND_BOTTOM, 0, 0, 0, SWP_NOSIZE | SWP_HIDE | SWP_REACTIVATE);
        // 3. Hide the window.
        pPage->ShowWindow(FALSE);
    }
}

//-----
BOOL CTabCrSheet::PreTranslateMessage(MSG* pMsg)
{
    // See what the current message is.
    switch (pMsg->message)
    {
        case WM_KEYDOWN:
        {
            case VK_RETURN:
                // Let the page handle commands.
                return GetActivePage()->DialogMessage(pMsg);
            case VK_TAB:
                if (GetKeyState(VK_CONTROL) < 0)
                {
                    // See if moving backward.
                    if (GetKeyState(VK_SHIFT) < 0)
                    {
                        if (GetActiveIndex() == 0)
                        {
                            SendMessage(WM_KEYDOWN, WPARAM)VK_END, 0;
                        }
                        else
                        {
                            SendMessage(WM_KEYDOWN, WPARAM)VK_LEFT, 0;
                        }
                    }
                }
            }
    }
}

```

TCSHEET.CPP 3-26-98 12:17p

```

else
{
    if (GetActiveIndex() == GetPageCount() - 1)
        SendMessage(WM_KEYDOWN, WPARAM)VK_END, 0;
    else
        SendMessage(WM_KEYDOWN, WPARAM)VK_RIGHT, 0;
    return TRUE;
}
break;
}
// Call the base class handler.
return CTabCrSheet::PreTranslateMessage(pMsg);
}

//-----
// CTabCrSheet message handlers
BEGIN_MESSAGE_MAP(CTabCrSheet, CTabCrSheet)
    //((AFX_MSG_MAP(CTabCrSheet)
    ON_NOTIFY_REFLECT(TON_SELCHANGE, OnSelChange)
    ON_NOTIFY_REFLECT(TON_SELCANCEL, OnSelCancel)
    //((AFX_MSG_MAP
    END_MESSAGE_MAP()

void CTabCrSheet::OnSelChange(WPARAM wParam, LPARAM lParam)
{
    // See if current page will deactivate.
    CPropertyPage* pPage = GetPage(GetActiveIndex());
    if (pPage->KillActive())
    {
        // Page deactivated.
        ActivatePage(pPage, FALSE);
        "Result = 0;
    }
    else
    {
        // Page refused to deactivate.
        "Result = 1;
    }
}

void CTabCrSheet::OnSelChange(WPARAM wParam, LPARAM lParam)
{
    // See if new page will activate.
    CPropertyPage* pPage = GetPage(GetActiveIndex());
    if (pPage->KillActive())
    {
        // Page activated.
        ActivatePage(pPage, TRUE);
        "Result = 0;
    }
    else
    {
        // Page refused to activate.
        "Result = 1;
    }
}

void CTabCrSheet::SetClientRect(CRect* pRect)
{
    m_rect = *pRect;
}

```

Page 3 of 4

```
OnPropertyChange "pPage = GetActivePage ();  
if (pPage != NULL)  
    ActivePage (pPage);  
}  
// TODO: Add your message handler code here  
}
```

```

// Since the dialog has been closed, return FALSE so that we exit the
// application, rather than start the application's message pump.
return FALSE;

}

// TUDO: Place code here to handle when the dialog is
// dismissed with OK
}

// TUDO: Place code here to handle when the dialog is
// dismissed with Cancel
}

}

// Since the dialog has been closed, return FALSE so that we exit the
// application, rather than start the application's message pump.
return FALSE;
}

```

[illegible]

Page 1 of 2


```

DialogBox_DlgProc(0);
}
else
{
    DialogBox(GetSystemCommand(HWND, 1, Param);
}
}

// If you add a minimize button to your dialog, you will need the code below
// to draw the icon. For MFC applications using the document/view model,
// this is automatically done for you by the framework.

void CDialog1_PDlg::OnPaint()
{
    if (!IsIconic())
    {
        CPaintDC dc(this); // device context for painting

        SendMessage(WM_IconSize, (WPARAM) dc.GetSafeHdc(), 0);

        // Center icon in client rectangle
        int cxIcon = GetSystemMetrics(SM_CX_ICON);
        int cyIcon = GetSystemMetrics(SM_CY_ICON);
        CRect rect;
        GetClientRect(&rect);
        int x = (rect.Width() - cxIcon + 1) / 2;
        int y = (rect.Height() - cyIcon + 1) / 2;

        // Draw the icon
        dc.DrawIcon(x, y, m_hIcon);
    }
    else
    {
        CDialog::OnPaint();
    }
}

// The system calls this to obtain the cursor to display while the user drags
// the minimized window.
HCURSOR CDialog1_PDlg::OnQueryDragIcon()
{
    return (HCURSOR) m_hIcon;
}

```

```
// this is the header file for the code to convert from variant to the numeric encoding
// that we internally call group. Note: this is different from the OED group which is
// like an ID number. Our group is an encoding of a name consisting of IPA characters
// into a numeric representation of which group of IPA characters a character in the
// name corresponds to. This is done using the grouparray.dat file which contains the
// group assignments for each valid IPA character. grouparray.dat is loaded into memory
// and is used as a lookup table.
```

```

UNIT :defined(AFX_CQWVAP2GRP_H_5D5A241_65D0_11D1_837A_000F820F4A_INCLUDED_)
#define AFX_CQWVAP2GRP_H_5D5A241_65D0_11D1_837A_000F820F4A_INCLUDED_

```

■if MSC VER >= 1000

program warning (double: 4786)

Monday 11 Mrs. V.

// conwar@corp.uh.edu

include «stream»

```
#include <use_ansi.h>
```

```
#include <iterator>
```

```
#include <map>
```

Received 15 June 2006; accepted 17 July 2006

...tion number 674.

.....

SECRET

10

public:

`ConvVar2Grp() :`

```
// Display Data
```

```
//PARALLELIZATION FOR CRP
enum { IDD = IDD_CONVERT VAR TO CRP };
```

```

CREDIT  m_base_errors_box,
CREDIT  a_base_ana_source_file_box

```

```

31341: m_gen_indices_vec_box;

```

```

C341t  m_gen_indexes_source_box;

```

```

CEdit  m_source_file_for_groups_box

```

```
CEdit m_sort_groups_source_box;
```

CEdit m desdupes group dest. box;

Created: 2013-03-27 10:10:10
 Updated: 2013-03-27 10:10:10
 Version: 1.0.0
 Author: [redacted]
 License: [redacted]

```

CREDIT  M_NUM_UNIQUE_VARIABLES_DONOR
CREDIT  M_NUM_UNIQUE_VARIABLES_DONOR
CREDIT  M_NUM_UNIQUE_VARIABLES_DONOR

```

```

CREDIT m_numgroups_read_in_box;

```

[illegible][illegible]

// Copyright (C) 1998, Language Analysis Systems Inc.

```
// the Defines.
#ifndef DEFINES_H
#define DEFINES_H

#ifndef MAX_IMAGE_SIZE
#define MAX_IMAGE_SIZE 30
#endif

#ifndef MAX_IMAGES_LIN
#define MAX_IMAGES_LIN 500
#endif

#ifndef FEATURE_DIST_FILE
#define FEATURE_DIST_FILE "feat.dat"
#endif

#ifndef DEF_IMAGES_FILE
#define DEF_IMAGES_FILE "names.txt"
#endif

#ifndef DEF_RULES_FILE
#define DEF_RULES_FILE "angl.rul"
#endif

#ifndef DEF_EXPANDED_IMAGES_FILE
#define DEF_EXPANDED_IMAGES_FILE "expanded_names.txt"
#endif

#ifndef DEF_IMAGES_FILE
#define DEF_IMAGES_FILE "names.namesec"
#endif

#endif
```

```

//class OkNameDialog : public CDialog
class OkNameDialog : public CPropertyPage
{
public:
    DECLARE_DYNCREATE(OkNameDialog)
    // standard constructor
    OkNameDialog() {}
    // Dialog Data
    // These are controls, most of which get us to the file name information from the user
    //({AFX_DIALOG(OkNameDialog)})
    enum { IDD = IDD_OKNAME_DIALOG };
    CEdit m_edit_names;
    CProgressCtrl m_pb_progress_bar;
    CEdit m_simplified_rules;
    CEdit m_processing_log;
    CEdit m_hispanic_variants;
    CEdit m_hispanic_names;
    CEdit m_chinese_variants;
    CEdit m_chinese_names;
    CEdit m_arabic_variants;
    CEdit m_arabic_names;
    CEdit m_english_variants;
    CEdit m_english_rules;
    CEdit m_chinese_rules;
    CEdit m_hispanic_rules;
    CEdit m_browse;
    CButton m_log;
    //({AFX_DIALOG(OkNameDialog)})
    //({AFX_VIRTUAL(OkNameDialog)})
protected:
    virtual void DoDataExchange(CDataExchange* pDX);
    //({AFX_VIRTUAL})
    // Implementation
public:
    // this file name string needs to be public to get the sorting done
    CString m_input_names;
    HICON m_hicon;
    // these CString objects hold all the various file names
    CString m_processing_log, m_hispanic_variants, m_hispanic_names,
    m_chinese_variants, m_chinese_names, m_arabic_names,
    m_english_variants, m_english_rules, m_arabic_rules,
    m_hispanic_rules, m_simplified_rules;
    char *mDir;
    // a nas object that will do name classification
    nas *mNas;
    // the four rule sets that will process names as classified *
    RuleSet *mRuleSet1, *mRuleSet2, *mRuleSet3, *mRuleSet4;
    // the active file name control, for Browse to affect
    // the Browse button will fill in contents for whichover
    // was last clicked into
    CEdit *mActiveFile;

```

Page 1 of 2

```

// use the version of these in parse.h
//define theVOWELS
//define is_VOWEL(ch)
//define *sinfo/|u|l/*
(srch(chVOWELS, ch) != NULL)

for the master declarations.

// vowel-intern variants codes. see ranker.h for the master declarations.
'5'
'N'
'A'

// 'standard' file extension indicator for browse file dialog box
// here tell the dialog box what file types to bring up by default
0
1
2
3
4

// The following section is required by PC-VAS.
//define RMS_SIZE MAX_LEN
//define KCH1MAXX
//include cme_and.h
//include ansp.
#pragma warning(disable: 4786)
using namespace std;
#include <iostream>
// end PC-VAS set up

// Ruleset class and related classes

```


TDSUT1-1.H 3-24-98 12:17p

10SUT1-2.H 3-24-98 12:17p


```

//[[[AF7_KSPICMAQr1Sheet]
atx_msg void GetExchangeTimeK* pMEMK, LRESULT* pResult);
atx_msg void GetExchangeTimeK* pMEMK, LRESULT* pResult);
//[[[AF7_KSP
DECLASS_MESSAGE_MAP()
}
//
#endif

```

PCNAS


```

// Copyright (C) 1998, Language Analysis Systems Inc.
//
// If defined(NFX_BATCH_H_A1984641_SF62_11D1_9189_0000624D5D9_INCLUDED_)
// define NFX_BATCH_H_A1984641_SF62_11D1_9189_0000624D5D9_INCLUDED_
//
// If _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000
// Batch.h : header file
//
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Chatting dialog
class CChatDlg : public CDialog
{
// Construction
public:
    CChatDlg(CWnd* pParent = NULL); // standard constructor
    CChatDlg(CWnd* p);

// Dialog Data
//{{AFX_DATA(CChatDlg)
enum { ID = IDD_BATCH_DIALOG };
CListBox m_results_list;
CEdit m_filedit;
//}}AFX_DATA

// Overrides
// ClassWizard generated virtual function overrides
//{{AFX_VIRTUAL(CChatDlg)
protected:
    virtual void DoDataExchange(CDataExchange* pDX); // DDX/DDV support
//}}AFX_VIRTUAL

// Implementation
protected:
    BOOL m_has;

// Generated message map functions
//{{AFX_MSG(CChatDlg)
afx_msg void OnButtonClick();
virtual void OnCancel();
virtual BOOL OnInitialDialog();
afx_msg void OnEdit();
//}}AFX_MSG
DECLARE_MESSAGE_MAP()
};

//{{AFX_INSERT_LOCATION}}
// Microsoft Developer Studio will insert additional declarations immediately before the previous line.

#endif // defined(NFX_BATCH_H_A1984641_SF62_11D1_9189_0000624D5D9_INCLUDED_)

```



```

// Copyright (C) 1998, Language Analysis Systems Inc.
//
// #ifndef NFX_PCNAS_H_03957A65_1861_11D1_93B9_0006E240509_INCLUDED_
// #define NFX_PCNAS_H_03957A65_1861_11D1_93B9_0006E240509_INCLUDED_
//
// #if _MSC_VER >= 1000
// #pragma once
// #endif // _MSC_VER >= 1000
//
// #ifndef _APPROX_H_
// #error Include "stdafx.h" before including this file for PCH
// #endif
//
// #include "resource.h" // main symbols
//
// .....
//
// #ifndef _DEBUG
// #define _DEBUG_STR " (Debug)"
// #else
// #define _DEBUG_STR ""
// #endif
//
// Version information
// #define VERSION_MAJOR 3
// #define VERSION_MINOR 1
//
// #define PROGRAM_NAME "PC Name Analysis System" _DEBUG_STR
// #define DEVELOP_STR "Copyright (C) 1998, Language Analysis Systems Inc."
// #define NAME2 "Individual-Name-Evaluation Utility."
//
// #define NAME_SIZE 16
//
// .....
//
// #ifndef _AFX_H_
// #error See pcnas.cpp for the implementation of this class
//
//
// class CPeNASApp : public CWinApp
// {
// public:
//     CPeNASApp();
//
// // Overrides
// // ClassWizard generated virtual function overrides
// #ifndef _AFX_VIRTUAL_
// public:
//     virtual BOOL InitInstance();
// #endif _AFX_VIRTUAL_
//
// // Implementation
//
// #ifndef _AFX_MFC_
// #error NFX_MFC(CPeNASApp)
// // NOTE - the ClassWizard will add and remove member functions here.
// // DO NOT EDIT what you see in these blocks of generated code !
// #endif _AFX_MFC_
//     DECLARE_MESSAGE_MAP()
// };
//
// .....
//
// #ifndef _AFX_INSERT_LOCATION_
// #error Microsoft Developer Studio will insert additional declarations immediately before the previous line.

```


[illegible]

Page 1 of 3

```
// password.cpp : Implementation file
//
// Copyright (C) 1998, Language Analysis Systems Inc.

#include "stdafx.h"
#include "passwd.h"

#define MAXMAX
#include "aws.h"
#include "map"

#pragma warning(disable: 4786)
using namespace std;

static_OBDS
define new OBDS_JOB
under THIS_FILE
static char THIS_FILE[] = __FILE__
endif

// =====
// ObOutDlg dialog used for App About
// =====

class CObOutDlg : public Dialog
{
public:
    CObOutDlg();

// Dialog Data
//{{AFX_DATA(CObOutDlg)
enum { IDO = IDD_AOUTBOX };
}}AFX_DATA

// ClassWizard generated virtual function overrides
//{{AFX_VIRTUAL(CObOutDlg)
protected:
    virtual void DoDataExchange(CDataExchange* pDX); // DDX/DDV support
//}}AFX_VIRTUAL

// Implementation
protected:
//{{AFX_MSG(CObOutDlg)
virtual BOOL OnInitDialog();
DECLARE_MESSAGE_MAP()
}}AFX_MSG

//{{AFX_PRIVATE(CObOutDlg)
private:
}}AFX_PRIVATE

void CObOutDlg::DoDataExchange(CDataExchange* pDX)
{
    Dialog::DoDataExchange(pDX);
}

BEGIN_MESSAGE_MAP(CObOutDlg, Dialog)
    ON_COMMAND(IDO, OnCommand)
END_MESSAGE_MAP()
```

PCNASDLG.CPP 3-24-98 1:11p


```

msg_str.Format("%.1f", m_nas->get_h_score());
mc_results_hispanic.SetWindowText(msg_str);

// put us back at the beginning
GoDlgCtrl(GetDlgItem(IDC_BTN_RESET));

}

void CPMASDlg::OnChangedEditThreshold()
{
    CString temp_str;
    mc_a_threshold.GetWindowText(temp_str);
    if (temp_str.GetLength() > 0) {
        // setting both since we don't have separate controls for these
        m_nas->set_ag_threshold(atof(temp_str));
        m_nas->set_aa_threshold(atof(temp_str));
    }
}

void CPMASDlg::OnSetFocusEditThreshold()
{
    mc_a_threshold.SetSel(0, -1, FALSE);
}

void CPMASDlg::OnChangedEditThreshold()
{
    CString temp_str;
    mc_c_threshold.GetWindowText(temp_str);
    if (temp_str.GetLength() > 0) {
        // setting both since we don't have separate controls for these
        m_nas->set_cg_threshold(atof(temp_str));
        m_nas->set_cs_threshold(atof(temp_str));
    }
}

void CPMASDlg::OnSetFocusEditThreshold()
{
    mc_c_threshold.SetSel(0, -1, FALSE);
}

void CPMASDlg::OnChangedEditThreshold()
{
    CString temp_str;
    mc_h_threshold.GetWindowText(temp_str);
    if (temp_str.GetLength() > 0) {
        // setting both since we don't have separate controls for these
        m_nas->set_hg_threshold(atof(temp_str));
        m_nas->set_hs_threshold(atof(temp_str));
    }
}

void CPMASDlg::OnSetFocusEditThreshold()
{
    mc_h_threshold.SetSel(0, -1, FALSE);
}

void CPMASDlg::OnLaunch()
{
    CPMASDlg batch_dialog(m_nas);
    batch_dialog.DoModal();
}

```

```

}

void CPMASDlg::OnKillFocusEditName()
{
    mc_name.SetWindowText(m_name);
}

void CPMASDlg::OnSetFocusEditName()
{
    mc_name.SetSel(0, -1, FALSE);
}

void CPMASDlg::OnChangedEditName()
{
    // TODO: If this is a RICHTEXT control, the control will not
    // send this notification unless you override the OnDialogBnDialog()
    // function to send the DLServWinHook message to the control
    // with the BM_CLICK flag fired into the lParam sub.
    m_name.MoveUp();
}

void CPMASDlg::OnAbout()
{
    CPMASDlg about_box;
    about_box.DoModal();
}

void CPMASDlg::OnAnalyze()
{
    if (m_nas->get_status() != 0)
        return;
    const char *RES_M52 = "The name is appears to be is.";
    CString msg_str;
    HICON icon;

    mc_results.SetWindowText("Waiting for analysis to complete...");
    switch (m_nas->analyze(m_name)) {
        case NAS_NAS_ARABIC:
            msg_str.Format(RES_M52, m_name, "Arabic");
            icon = MFCApp() ->LoadIcon(IDI_FLAGARABIC);
            break;
        case NAS_NAS_CHINESE:
            msg_str.Format(RES_M52, m_name, "Chinese");
            icon = MFCApp() ->LoadIcon(IDI_FLAGCHINA);
            break;
        case NAS_NAS_HISPANIC:
            msg_str.Format(RES_M52, m_name, "Hispanic");
            icon = MFCApp() ->LoadIcon(IDI_FLAGSPAIN);
            break;
        case NAS_NAS_URDU:
            msg_str.Format(RES_M52, m_name, "Urdu");
            icon = MFCApp() ->LoadIcon(IDI_FLAGURDU);
            break;
        default:
            break;
    }
    mc_results.SetWindowText(msg_str);
    mc_results.Icon.SetIcon(icon);

    msg_str.Format("%.1f", m_nas->get_a_score());
    mc_results_arabic.SetWindowText(msg_str);
    msg_str.Format("%.1f", m_nas->get_c_score());
    mc_results_chinese.SetWindowText(msg_str);

```



```

//({{ID,DEPSUBJECTS}}
// Microsoft Developer Studio generated include file.
// Used by pcma.ac
//
// Copyright (C) 1998, Language Analysis Systems Inc.

#define IDM_ABOUTBOX 0x0010
#define ID_ABOUTBOX 100
#define ID_ABOUTBOX 101
#define ID_POWERS_DIALOG 102
#define ID_MAILMERGE 128
#define ID_FUSPAIN 130
#define ID_RUARK 131
#define ID_DIALOG 132
#define ID_DIALOG 133
#define ID_BATCH_DIALOG 140
#define ID_EDIT_RANGE 1000
#define ID_STATIC_HEADER 1002
#define ID_ANALYZE 1003
#define ID_RANGE_RESULTS 1004
#define ID_RESULTS_ARABIC 1005
#define ID_RESULTS_CHINESE 1006
#define ID_RESULTS_HISPANIC 1007
#define ID_ABOUT 1009
#define ID_EDIT_A_THRESHOLD 1011
#define ID_EDIT_C_THRESHOLD 1012
#define ID_EDIT_H_THRESHOLD 1013
#define ID_TAB1 1014
#define ID_VERSION 1015
#define ID_VERSION2 1016
#define ID_RESULTS_CON 1017
#define ID_STATIC_ID 1018
#define ID_BATCH 1020
#define ID_BATCH_FILE 1021
#define ID_BATCH_BROWSER 1022
#define ID_SUBMIT 1026
#define ID_RESULTS_LIST 1031

// Next default values for new objects
//
#define APSTUDIO_INVOKED
#define APSTUDIO_READONLY_SYMBOLS
#define APP_NEXT_RESOURCE_VALUE 141
#define APP_NEXT_COMMAND_VALUE 12771
#define APP_NEXT_CONTROL_VALUE 1012
#define APP_NEXT_SYMBD_VALUE 105
#endif
#endif

```

```
// stdafx.cpp : source file that includes just the standard includes
// pmas.pch will be the pre-compiled header
// stdafx.obj will contain the pre-compiled type information
// Copyright (C) 1998, Language Analysis Systems Inc.

#include "stdafx.h"
```


PCNASLIB


```
def.cpp: implementation of the def class.
```

```
// Only attempt to open file stream if it is not opened.
if (!fs.isopen()) {
    rc = false;
    while (true) {
        inc i = 0;
        do {
            switch (access_model) {
                case DE_AAC_MODE_READ :
                    fs.open(name, ios::in | ios::binary /*| ios::nocreate*/);
                    break;
                case DE_AAC_MODE_READ_STREAMS :
                    fs.open(name, ios::in | ios::binary /*| ios::nocreate, filebu
                        break;
                case DE_AAC_MODE_READ_STREAM :
                    fs.open(name, ios::in | ios::binary /*| ios::nocreate, filebu
                        break;
                case DE_AAC_MODE_READ_WRITE :
                    fs.open(name, ios::in | ios::out | ios::binary /*| ios::nocre
                        break;
                case DE_AAC_MODE_READ_WRITE_STREAMS :
                    fs.open(name, ios::in | ios::out | ios::binary /*| ios::nocre
                        break;
                default: // is read/write
                    fs.open(name, ios::in | ios::out | ios::binary /*| ios::nocre
            }
            rc = fs.good();
            if (!rc && retry_count > 0) {
                // Addressing an error msg(CB12, name)). // Recrying to "waiting for a
                Sleep(retry_delay);
                fs.close(); // this close() really is needed
            }
            while ((rc && ++i < retry_count);
            if (!rc && retry_count > 0) {
                // If Addressing an error msg(13, name, i * retry_delay / 1000.0), NO_RETRY
                // The input file is was unavailable for lg seconds.\Who you want to t
                //break; // from the while loop
            }
            break; // from the while loop
        } else // all OK or only one try allowed
        } // while
    } // if not open
    return rc;
} // reopen_file

uint def::get_field_info(void)
{
    def_field fld;
    if (!fs.good()) {
        b.bad = true;
        return EXIT_FAILURE;
    }
    fs.read((char *)bhead, sizeof(head));
    if (!fs.good()) {
```

```

        b_bad = true;
        return EXIT_FAILURE;
    }

    // use record length to allocate rec_buf
    // try {
    //     rec_buf = new char(head_rec_size + 1);
    // }
    // catch (x_alloc) {
    //     DF_SHOW_MESSAGE MSG, "Unable to allocate rec_buf in vidb_1().";
    //     //exit(EXIT_FAILURE);
    // }

    // find out how many fields there are and allocate the f_list array
    f_count = (int) (head_data_offset - fs.tell()) / sizeof(fid);
    f_list = new dbf_field[unsigned int] f_count;

    // read each field record and store it (field offset starts at 1
    // in order to skip the deleted flag at the beginning of each record
    word
    short int n;
    fidoff = 1;
    for (n = 0, fidoff = 1; n < f_count; n++) {
        fs.read( (char *) &fid, sizeof(fid) );
        if (ifs.good()) break;
        // check end of field list delimiter word
        if ( fid.name(0) == 0x0d ) break;
        memcpy(&f_list[n], &fid, sizeof(fid));
        f_list[n].set_off(fidoff);
        fidoff = (word)fidoff + (word)fid.length();
    }

    if (ifs.good()) || b_bad)
        return EXIT_FAILURE;

    // jump to the data (should already be there)
    // should we check if fs.seekg(0, ios::beg).tell() == 0?
    fs.seekg( (long)head_data_offset, ios::beg);

    return EXIT_SUCCESS;
} // get_field_info

void dbf::setup(void)
{
    rec_buf = NULL;
    f_list = NULL;
    b_bad = false;
    b_eof = false;
    current_record = 1;
}

char *dbf::read_buf(void)
{
    // seek to start of current record
    fs.seekg(head_data_offset + (current_record - 1) * (int)head_rec_size,
        ios::beg);
    if (ifs.good())
        return NULL;
}

```

DBF.CPP 3-24-98 1:11p

```

// read the record into rec_buf
fs.read(rec_buf, (int)head_rec_size);
if (ifs.good() || fs.gcount() == 0)
    return NULL;

rec_buf[head_rec_size] = 0x0; // terminate da record

// repeat line from above in order to leave file pointer at start of record
fs.seekg(head_data_offset + (current_record - 1) * (int)head_rec_size,
    ios::beg);
if (ifs.good())
    return NULL;

return rec_buf;
}

// skip changes the current record variable and returns the number of
// records actually skipped
long dbf::skip(long lskip)
{
    unsigned long oldrec = current_record;
    current_record += lskip;

    b_eof = false;
    if ( current_record > head_last_rec ) {
        b_eof = true;
        current_record = head_last_rec;
    }
    else if (current_record == 0)
        current_record = 1;

    return current_record - oldrec;
}

// the position file_ptr defaults to false, which means that only
// the record number will be changed. When the value is true, the
// file pointer is modified as well.
bool dbf::go_to(unsigned long lrec, bool position, file_ptr)
{
    if ( current_record > head_last_rec ) {
        b_eof = true;
        current_record = head_last_rec;
        return false;
    }
    else if (0 < lrec && lrec <= head_last_rec) {
        b_eof = false;
        current_record = lrec;
        if (position, file_ptr)
            fs.seekg(head_data_offset + (current_record - 1) * (int)head_rec_size,
                ios::beg);
        return true;
    }
    else
        return false;
}

int dbf::pos(const char *fname)
{
    for (int i = 0; i < f_count; i++)
        if (strcmp(fname, f_list(i).field_name) == 0)
            return i;
    return -1;
}

```

Page 2 of 3

```

    }

    int dbf::offset(const char *field)
    {
        int p;
        p = pos(field);
        if (p < 0)
            return -1;
        return f_list[p].offset();
    }

    int dbf::offset(int p) const
    {
        return f_list[p].offset();
    }

    int dbf::length(const char *field)
    {
        int p;
        p = pos(field);
        if (p < 0)
            return 0;
        return f_list[p].length();
    }

    int dbf::length(int in_pos) const
    {
        return f_list[in_pos].length();
    }

    unsigned long dbf::record(void)
    {
        return current_record;
    }

    bool dbf::inc_last_rec(void)
    {
        if ( this->bad() || (fs.good() == false) )
            return false;
        head.last_rec++;
        if ( this->bad() || (fs.good() == false) )
            return false;
        fs.seek( 0L, ios::beg );
        if ( this->bad() || (fs.good() == false) )
            return false;
        fs.write( (char *)head, sizeof(head) );
        if ( this->bad() || (fs.good() == false) )
            return false;
        fs.flush();
        if ( this->bad() || (fs.good() == false) )
            return false;
        return true;
    }

    unsigned long dbf::get_last_rec(void) const
    {
        return head.last_rec;
    }

```

```

word dbf_field::length(void) const
{
    switch (field_type) {
        case 'C':
            return len_info.char_len;
        case 'L':
            return 1;
        case 'D':
            return 8;
        case 'M':
            return 10;
        case 'W':
            return (word)len_info.max_size_len;
        default:
            return 0;
    }
}

/* ..... */
/* ..... */

```

```
dbf.h: interface for the dbf class.
```

```

#include <db.h>
#include <db.h>
//if _MSC_VER >= 1000
#include <io.h>
#include <fcntl.h>
#include <windows.h>
using namespace std;

// Define some standard modes for accessing the database.
// We need a way to make sure that files that are read only
// get opened as read only, otherwise, we will fail to open
// the file if it is on a CD-ROM.
enum db_acc_mode_t {
    DB_ACC_MODE_READ_WRITE,
    DB_ACC_MODE_READ,
    DB_ACC_MODE_READ_WRITE_SHRINK,
    DB_ACC_MODE_READ_SHRINK,
    DB_ACC_MODE_READ_SHRINK
};

```

```

class dbf_header
{
public:
    byte    dbf_id;
    byte    last_update();
    unsigned long last_rec;
    word    data_offset;
    word    rec_size;

private:
    byte    dummy[20];
};

```

```

/* dbf_field
A field class representing a DBF field record
the dummy fields are left in so the class data elements
can be read directly from the file

```

DBF.H 3-24-98 1:11p

```

class dbf_field
{
public:
    char field_name[11];
    byte field_type;

    word offset_pos; // NOTE: we will be using part of
    char dummy[2]; // this dummy area to hold the field's
    // offset; was: char dummy[4];

    union // the following bytes have two
    { // different meanings depending
        word char_len; // on the type of the field
        struct {
            byte len;
            byte dec;
        } len_info;
    } len_info;

    char dummy2[14]; // leave dummy areas so the class
    // can be read from a DBF file

public:
    char name() const { return field_name; } // it's null terminated
    byte type() const { return field_type; }
    void set_offset_pos( int offset_pos = 0 ) {
        word offset() const { return offset_pos; }
        word length() const;
    }
    byte decimals() const
    {
        return type() == 'N' ? len_info.len_info.dec : (byte)0;
    }
};

/* dbf - database class */
class dbf
{
public:
    int f_count;
    void setup();

protected:
    char name[ MAX_PATH + 1 ];

    bool b_bad;
    bool b_eof;
    int access_mode;
    fstream fs;
    dbf_header head;
    char *rec_buf;

public:
    dbf_field *f_list; // this should not be public!

    // info about the current record
    unsigned long current_record;
    bool deleted;

    int get_field_info();
    bool open_file(const char *file, int acc_mode, int retry_count, DWORD retry_delay);
    bool reopen_file(int retry_count, DWORD retry_delay);
    void close_file();

```

Page 1 of 2

```

protected:
    bool inc_last_rec();

public:
    dbf(const char *filename, db_acc_mode_t access_mode, int retry_count, DBD_RETRY_DELAY);
    virtual ~dbf();

    unsigned long get_last_rec() const;
    char *get_file_name() { return name; }
    bool bad() const { return !is_good() || b_bad; }
    bool db_eof() const { return b_eof; }

    virtual char *read_buf(); // read current record into rec_buf
    virtual char *db_read()
    {
        (void) read_buf();
        return rec_buf;
    }

    long skip(long = 1UL);
    bool go_to(unsigned long lrec, bool position_file_per = FALSE);

    int pos(const char *);
    int offset(const char *field);
    int offset(int position) const;
    int length(const char *field);
    int length(int position) const;

    int get_f_count() const { return f_count; }
    unsigned long record();

    void flush() { (void) fs.flush(); }

};
/* ..... */
/* ..... */
#endif

```



```

it = t.find(d); \
if (it != t.end()) \
    s += (*it).second; \
}

/* analyze
Output: Returns a guess of the language type.
Input: A terminated character array. Loading and trailing blanks will
be removed.
*/
const nas::language nas::analyze(const char *in_name)
{
    m_name(0) = '\0';
    strcat(m_name, in_name);
    strstrip(m_name, " \t");
    strstrip(m_name);

    // Before getting into the n-gram counting, see if the name
    // has already been defined in one of the tables.
    // No name should be in more than one table!
    name_table_t::iterator it;
    bool table_hit = false;

    m_a_score = 0.0;
    m_c_score = 0.0;
    m_h_score = 0.0;

    use_names(m_arabic_g_names, m_name, m_a_score);
    if (!table_hit) use_names(m_arabic_g_names, m_name, m_a_score);
    if (!table_hit) use_names(m_chinese_g_names, m_name, m_c_score);
    if (!table_hit) use_names(m_chinese_g_names, m_name, m_c_score);
    if (!table_hit) use_names(m_hispanic_g_names, m_name, m_h_score);
    if (!table_hit) use_names(m_hispanic_g_names, m_name, m_h_score);

    if (!table_hit) {
        // These values must come from the way the statistics were gathered
        // Note that we have effectively disabled the given-name stuff until
        // the files are created properly.
        double t_ag = 2001.0, t_aa = 100.0;
        double t_cg = 2001.0, t_ca = 100.0;
        double t_hg = 2001.0, t_ha = 100.0;

        // Calculate the digraph scores.
        char *p;
        WORD w;
        digraph_table_t::iterator it;
        for (p = m_name; *p != EOS && (*p != '\0' || *p != EOS); p++) {
            w = (*p) << 8 | (*p + 1);

            use_ngrams(m_arabic_g_names, w, t_ag);
            use_ngrams(m_arabic_g_names, w, t_aa);
            use_ngrams(m_chinese_g_names, w, t_cg);
            use_ngrams(m_chinese_g_names, w, t_ca);
            use_ngrams(m_hispanic_g_names, w, t_hg);
            use_ngrams(m_hispanic_g_names, w, t_ha);
        }

        // Calculate the trigraph scores (using only the first and last trigraphs).
        size_t name_len;
        if ((name_len = strlen(m_name)) >= 4) { // The #, #, and two actual letters
            WORD tri_str;
            trigraph_table_t::iterator it;
            char *p;
            p = m_name;
            tri_str = (*p << 16) + ((*p + 1) << 8) + (*p + 2);
        }
    }
}

```

```

digraph_table_t *g_which, *s_which;

switch (lang) {
case NAS_ARABIC:
    g_which = m_arabic_g_names;
    s_which = m_arabic_s_names;
    break;
case NAS_CHINESE:
    g_which = m_chinese_g_names;
    s_which = m_chinese_s_names;
    break;
case NAS_HISPANIC:
    g_which = m_hispanic_g_names;
    s_which = m_hispanic_s_names;
    break;
default:
    return 21;
    break;
}

rc = load_digraphs(lang, di_g_name, g_which);
if (rc == 0)
    rc = load_digraphs(lang, di_s_name, s_which);
// Load digraph tables

if (rc == 0) { // Load trigraph tables
    trigraph_table_t *g_which, *s_which;

    switch (lang) {
    case NAS_ARABIC:
        g_which = m_arabic_g_names;
        s_which = m_arabic_s_names;
        break;
    case NAS_CHINESE:
        g_which = m_chinese_g_names;
        s_which = m_chinese_s_names;
        break;
    case NAS_HISPANIC:
        g_which = m_hispanic_g_names;
        s_which = m_hispanic_s_names;
        break;
    default:
        return 31;
        break;
    }

    rc = load_trigraphs(lang, tri_g_name, g_which);
    if (rc == 0)
        rc = load_trigraphs(lang, tri_s_name, s_which);
    // Load trigraph tables

    return rc;
} // Init

// Two macros to make life easier in the next routine.
#define use_names(t, n, s) \
{ \
    it = t.find(n); \
    if (it != t.end()) { \
        s += (*it).second; \
        table_hit = true; \
    } \
}

#define use_ngrams(t, d, s) \
{ \
    it = t.find(d); \
    if (it != t.end()) { \
        s += (*it).second; \
    } \
}

```

```

use_sysname(m_arabic_g_trigraphs, tri_str, t_ag);
use_sysname(m_arabic_g_trigraphs, tri_str, t_ag);
use_sysname(m_chinese_g_trigraphs, tri_str, t_cg);
use_sysname(m_chinese_g_trigraphs, tri_str, t_cg);
use_sysname(m_chinese_g_trigraphs, tri_str, t_cg);
use_sysname(m_hispanic_g_trigraphs, tri_str, t_hg);
use_sysname(m_hispanic_g_trigraphs, tri_str, t_hg);
use_sysname(m_hispanic_g_trigraphs, tri_str, t_hg);

p = m_name + name_len - 1;
tri_str = (*p << 16) + ((*p << 1) << 8) + (*p << 2);

use_sysname(m_arabic_g_trigraphs, tri_str, t_ag);
use_sysname(m_arabic_g_trigraphs, tri_str, t_ag);
use_sysname(m_chinese_g_trigraphs, tri_str, t_cg);
use_sysname(m_chinese_g_trigraphs, tri_str, t_cg);
use_sysname(m_chinese_g_trigraphs, tri_str, t_cg);
use_sysname(m_hispanic_g_trigraphs, tri_str, t_hg);
use_sysname(m_hispanic_g_trigraphs, tri_str, t_hg);
use_sysname(m_hispanic_g_trigraphs, tri_str, t_hg);

}

a_s_score = MAX(t_ag, t_ag);
m_c_score = MAX(t_cg, t_cg);
m_h_score = MAX(t_hg, t_hg);

}

language rc = INS_UNKNOWN;

// Choose the largest score which is over the threshold.
// But that is ambiguous since there are several thresholds. So, do
// first the largest threshold, then see if that is over its threshold.
// Or do find those which are over their own thresholds, then take the
// largest of those? The former is being used right now.
// With only three languages, this method is just fine, but if more
// languages are added, this portion of code will have to be rewritten.
if ((m_s_score >= m_ag_threshold || m_s_score >= m_as_threshold)
    && m_s_score >= m_c_score && m_s_score >= m_h_score)
{
    rc = INS_ARABIC;
}
else if ((m_c_score >= m_cg_threshold || m_c_score >= m_cs_threshold)
    && m_c_score >= m_s_score && m_c_score >= m_h_score)
{
    rc = INS_CHINESE;
}
else if ((m_h_score >= m_hg_threshold || m_h_score >= m_hs_threshold)
    && m_h_score >= m_s_score && m_h_score >= m_c_score)
{
    rc = INS_HISPANIC;
}

return rc;
} // analyze

// Load name tables
int nas::load_names(const char *fname, name_table_t *which)
{
    const char *field_name1 = "NAME";
    const char *field_name2 = "SCORE";
    const int FID2_MID = 8;
    int rc;
    name_table_t::iterator nameTableIter;

    dbf table(fname, DB_ACC_RDONLY, 0, 0);
    if (table.bdf())
        rc = 12; // a name table was not accessible
    else {
        rc = 0;
        int field_off1 = table.offset(field_name1);
        int field_len1 = table.length(field_name1);

```

NAS.CPP 3-26-98 1:11p

```

int field_off2 = table.offset(field_name2);
int field_len2 = table.length(field_name2);

// Be wary about people changing the tables
if (field_off1 < 0)
    rc = 13;
else if (field_off2 < 0)
    rc = 14;
else if (field_len1 != NAS_NAME_SIZE)
    rc = 15;
else if (field_len2 != FID2_MID)
    rc = 16;
else {
    double n;
    name_t st, temp_st;
    const st(FID2_MID + 1);
    temp_st(FID2_MID + 1);
    temp_st(FID2_MID + 1);
    st(FID2_MID + 1);
    temp_st(FID2_MID + 1);
    while (!table.eof()) {
        char *buf = table.read(buf);
        strcpy(temp_st, buf + field_off1, field_len1);
        st.atrcis(st + 1, temp_st);
        strcpy(st, temp_st);
        n = atof(st);
        // add the name to the map, but first make sure
        // the name was not already there. Otherwise we
        // will get a memory leak, because strcpy will get
        // called, but the string will not get added to the map.
        nameTableIter = which->find(st);
        if (nameTableIter == which->end())
            (*which)[strcpy(st)] = n;
        table.skip();
    }
}

return rc;
} // load_names

// Load digraph tables
int nas::load_digraphs(const language lang,
    const char *fname, digraph_table_t *which)
{
    int rc = 0;
    const int FID2_MID = 8;
    const char *field_name1;
    const char *field_name2;

    // I wish people had simply chosen the same name for corresponding fields.
    switch (lang) {
        case NAS_ARABIC:
            field_name1 = "DI";
            field_name2 = "ALSCORE";
            break;
        case NAS_CHINESE:
            field_name1 = "DIGRAPH";
            field_name2 = "CScore";
            break;
        case NAS_HISPANIC:
            field_name1 = "DIGRAPH";
            field_name2 = "HSCORE";
            break;
    }
}

```

Page 3 of 6


```

        field_name1 = "TRIGRAPH";
        field_name2 = "SCORES";
        break;
    default:
        return 31;
        break;
    }

    def table(fname, DB_ACC_MODE_READ, 0, 0);

    if (table.buf(1))
        rc = 32; // a table was not accessible
    else {
        rc = 0;
        line_field_off1 = table.offset(field_name1);
        line_field_len1 = table.length(field_name1);
        line_field_off2 = table.offset(field_name2);
        line_field_len2 = table.length(field_name2);
        // be nasty about people changing the tables
        if (field_off1 < 0)
            rc = 33;
        else if (field_off2 < 0)
            rc = 34;
        else if (field_len1 != 3)
            rc = 35;
        else if (field_len2 != FLD2_WID)
            rc = 36;
        else {
            double n;
            double dw;
            char ss[FLD2_WID + 1];
            ss[FLD2_WID] = EOS;
            table.go_to(0); // set eof flag
            while (!table.db_eof()) {
                char *buf = table.read_buf(1);
                // pack the trigraph into a four-byte word
                dw = ((*buf + field_off1) << 16)
                    + ((*buf + field_off1 + 1) << 8)
                    + (*buf + field_off1 + 2);
                strcpy(ss, buf + field_off2, field_len2);
                n = atof(ss);
                (which) [dw] = n;
                table.skip(1);
            }
        }
        return rc;
    } // load_trigraphs
}

/* ..... */
const int mas::batch_testing(const char *fin_name, const char *fout_name,
                             const char *field_name)
{
    static char buf[BUFSIZ];
    static const char *lang_names[] = {
        "Arabic", "Chinese", "Hispanic", "Unknown"
    };
    if (get_status() != 0)
        return 81;
    FILE *f_in;
    if (strlen(fin_name) <= 0 || (f_in = fopen(fin_name, "rt")) == NULL)
        return 81;
}

```

Page 4 of 6

```

    default:
        return 31;
        break;
    }

    def table(fname, DB_ACC_MODE_READ, 0, 0);

    if (table.buf(1))
        rc = 22; // a table was not accessible
    else {
        rc = 0;
        line_field_off1 = table.offset(field_name1);
        line_field_len1 = table.length(field_name1);
        line_field_off2 = table.offset(field_name2);
        line_field_len2 = table.length(field_name2);
        // be nasty about people changing the tables
        if (field_off1 < 0)
            rc = 33;
        else if (field_off2 < 0)
            rc = 34;
        else if (field_len1 != 3)
            rc = 35;
        else if (field_len2 != FLD2_WID)
            rc = 36;
        else {
            double n;
            double w;
            char ss[FLD2_WID + 1];
            ss[FLD2_WID] = EOS;
            table.go_to(0); // set eof flag
            while (!table.db_eof()) {
                char *buf = table.read_buf(1);
                // pack the digraph into a two-byte word
                w = ((*buf + field_off1) << 8) + (*buf + field_off1 + 1);
                strcpy(ss, buf + field_off2, field_len2);
                n = atof(ss);
                (which) [w] = n;
                table.skip(1);
            }
        }
        return rc;
    } // load_digraphs
}

// Load trigraph tables
int mas::load_trigraphs(const language lang,
                        const char *fname, trigraph_table_t *which)
{
    int rc = 0;
    const int FLD2_WID = 8;
    char *field_name1;
    char *field_name2;

    // I wish people had simply chosen the same name for corresponding fields.
    switch (lang) {
        case MAS_ARABIC:
            field_name1 = "TRIGRAPH";
            field_name2 = "SCORES";
            break;
        case MAS_CHINESE:
            field_name1 = "TRIGRAPH";
            field_name2 = "SCORES";
            break;
        case MAS_HISPANIC:
            field_name1 = "TRIGRAPH";
            field_name2 = "SCORES";
            break;
    }
}

```

MAS-CPP 3-24-98 1:11p


```

/* 48 */ "NO SUCH ERROR",
/* 49 */ "NO SUCH ERROR",
// 50-51: reserved for X errors
/* 50 */ "NO SUCH ERROR",
/* 51 */ "NO SUCH ERROR",
/* 52 */ "NO SUCH ERROR",
/* 53 */ "NO SUCH ERROR",
/* 54 */ "NO SUCH ERROR",
/* 55 */ "NO SUCH ERROR",
/* 56 */ "NO SUCH ERROR",
/* 57 */ "NO SUCH ERROR",
/* 58 */ "NO SUCH ERROR",
/* 59 */ "NO SUCH ERROR",
// 60-69: reserved for X errors
/* 60 */ "NO SUCH ERROR",
/* 61 */ "NO SUCH ERROR",
/* 62 */ "NO SUCH ERROR",
/* 63 */ "NO SUCH ERROR",
/* 64 */ "NO SUCH ERROR",
/* 65 */ "NO SUCH ERROR",
/* 66 */ "NO SUCH ERROR",
/* 67 */ "NO SUCH ERROR",
/* 68 */ "NO SUCH ERROR",
/* 69 */ "NO SUCH ERROR",
// 70-79: reserved for X errors
/* 70 */ "NO SUCH ERROR",
/* 71 */ "NO SUCH ERROR",
/* 72 */ "NO SUCH ERROR",
/* 73 */ "NO SUCH ERROR",
/* 74 */ "NO SUCH ERROR",
/* 75 */ "NO SUCH ERROR",
/* 76 */ "NO SUCH ERROR",
/* 77 */ "NO SUCH ERROR",
/* 78 */ "NO SUCH ERROR",
/* 79 */ "NO SUCH ERROR",
// 80-89: reserved for batch-testing errors
/* 80 */ "Unknown language",
/* 81 */ "Unable to use MAS object",
/* 82 */ "Unable to open input file",
/* 83 */ "Incorrect name field offset",
/* 84 */ "Incorrect name field length",
/* 85 */ "NO SUCH ERROR",
/* 86 */ "NO SUCH ERROR",
/* 87 */ "NO SUCH ERROR",
/* 88 */ "Unable to open output file",
/* 89 */ "NO SUCH ERROR",
},

```

```

const char *mas_get_message(const int msg_code) const
{
    if (0 <= msg_code && msg_code < sizeof error_msg / sizeof error_msg[0])
        return error_msg[msg_code];
    else
        return "Unknown message code";
}
/* ..... */
/* ..... */
/* ..... */

```

nas.h: interface for the nas class.

Summer 1997: Robert Trubek - ported from the original Clipper code.

1. General:

This file defines the nas (Name-Analysis System) class, whose goal is to identify a name segment (no internal blanks allowed) as belonging to one of a limited set of language types; the set currently consists of Arabic, Chinese, and Hispanic names. Names for which the confidence level is low for all of these categories is identified as "unknown".

The classification algorithm is based upon statistical analysis of digraph and trigraph frequencies of known names. But a set of tables of well-known names is also used to assign scores to names before the counting phase and the scores will be taken from these tables when so found. A new name only appears in one table. The scores for the other languages will be assigned zero if the name is found in a table.

2. Interface:

The class constructor is to be given the names of the name, digraph and trigraph databases (.DBF files). The get_status() member function should then be checked immediately for a zero (0) value which would indicate satisfactory initialization; non-zero values would indicate specific reasons for failure. The function get_message() will return a string representation of the failure, if desired.

After construction, a name (a properly terminated character array) is passed to the analyze() member function, which returns one of the "language" enumerated types. Analyze() also returns the internal values of the calculated scores for each language group, and these are accessible through the get_a_score(), get_c_score() and get_h_score() member functions. Examining these values could yield more detailed information for further classification.

Besides having the highest score, the returned name category must also pass a threshold specific to each name category. These values can be set and gotten through the set_ag_threshold(), set_cg_threshold(), set_hg_threshold(), get_ag_threshold(), get_cg_threshold(), get_hg_threshold(), set_as_threshold(), set_cs_threshold(), set_hs_threshold(), get_as_threshold(), get_cs_threshold(), and get_hs_threshold() member functions.

The following lines are probably the minimum required in the calling module in order to use this header file:

```
#define KEMINMAX
#include <ust_ansi.h>
#include <omp>

#pragma warning(disable: 4786)
using namespace std;
```

3. Internals:

The DBF files are read using the dbf class defined elsewhere. The relevant contents are then stored into private STL map (name_table_t, digraph_table_t and trigraph_table_t) data members. The init() member

function is where all of this work is accomplished.

The keys for the maps are either a char pointer (char *), an unsigned two-byte quantity (a WORD) or an unsigned four-byte quantity (a DWORD). For digraphs, the two-byte key is used; for trigraphs, the three least-significant bytes are used. Appropriate bit shifting operations are used to transfer characters into these keys.

4. Tables (external files)

For each language there are three types of tables, NAME, DIGRAPH, and TRIGRAPH, and for each type there is one for given name and one for surname; hence, a total of 18 tables. The naming conventions should be clear from the sample Arabic tables:

ASMAE.DBF, NSI.DBF, ASTRI.DBF

The column names in the digraph and trigraph were unfortunately not well chosen when the original Clipper program was built, but we are retaining them here for backwards compatibility.

- * N.B. Currently (Summer 1997) the given-name digraph and trigraph capability is disabled since the tables themselves have not yet been built.

----- */

```
#ifndef NAS_H
#define NAS_H

#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000
```

```
// Added by ETS so that the code can compile on its own as a library
// Note that this means that NAS will (as currently set up), only
// consider the first 16 chars of a name when doing classification.
// Changes in this value might also require changes in the length
// of the associated DBF files, since their length currently is 16.
#define NAS_NAME_SIZE 16
```

```
class nas
{
public:
    enum language { NAS_ARABIC = 0, NAS_CHINESE, NAS_HISPANIC, NAS_UNKNOWN };

private:
    typedef char name_t[NAS_NAME_SIZE + 2 + 1]; // 2 chars for # and #
```

```
    // copy of name currently being analyzed
    name_t m_name;
    // scores being calculated
    double m_a_score;
    double m_c_score;
    double m_h_score;
    // threshold against which to compare scores
    double m_ag_threshold, m_as_threshold;
    double m_cg_threshold, m_cs_threshold;
    double m_hg_threshold, m_hs_threshold;
    // status of the analysis--zero means all OK
    int m_status;
```

```
    // needed for char* comparisons in the name maps
    template<class _Ty>
```

```

struct less_str : binary_function<Ty, Ty, bool> {
    bool operator()(const Ty& x, const Ty& y) const
    { return strcmp(x, y) < 0; }
};

typedef map< char *, double, less_str> char_* > name_table_t;
name_table_t m_arabic_g_names;
name_table_t m_chinese_g_names;
name_table_t m_hispanic_g_names;
name_table_t m_arabic_s_names;
name_table_t m_chinese_s_names;
name_table_t m_hispanic_s_names;

typedef map< WORD, double > digraph_table_t;
digraph_table_t m_arabic_g_digraphs;
digraph_table_t m_chinese_g_digraphs;
digraph_table_t m_hispanic_g_digraphs;
digraph_table_t m_arabic_s_digraphs;
digraph_table_t m_chinese_s_digraphs;
digraph_table_t m_hispanic_s_digraphs;

typedef map< DWORD, double > trigraph_table_t;
trigraph_table_t m_arabic_g_trigraphs;
trigraph_table_t m_chinese_g_trigraphs;
trigraph_table_t m_hispanic_g_trigraphs;
trigraph_table_t m_arabic_s_trigraphs;
trigraph_table_t m_chinese_s_trigraphs;
trigraph_table_t m_hispanic_s_trigraphs;

int init(const language lang)
{
    const char *n_g_fname, const char *d1_g_fname, const char *tri_g_fname;
    const char *n_s_fname, const char *d1_s_fname, const char *tri_s_fname;

    int load_names(const char *fname, name_table_t *which);
    int load_digraphs(const language lang,
                     const char *fname, digraph_table_t *which);
    int load_trigraphs(const language lang,
                      const char *fname, trigraph_table_t *which);

    void nls_delete_and_free_table_data(name_table_t *nTable);

public:
    nls(const language language)
    {
        const char *n_g_fname, const char *d1_g_fname, const char *tri_g_fname;
        const char *n_s_fname, const char *d1_s_fname, const char *tri_s_fname;
        const language language2;
        const char *n_g_fname2, const char *d1_g_fname2, const char *tri_g_fname2;
        const char *n_s_fname2, const char *d1_s_fname2, const char *tri_s_fname2;
        const language language3;
        const char *n_g_fname3, const char *d1_g_fname3, const char *tri_g_fname3;
        const char *n_s_fname3, const char *d1_s_fname3, const char *tri_s_fname3;
    }

    nls();

    const language analyze(const char *);
    const int batch_testing(const char *, const char *, const char *);
    const char *get_message(const int) const;

    // Zero (0) indicates all is well; non-zeros indicate specific error conditions.
    // Values in the tens indicate errors in the name tables.
    // Values in the twenties indicate errors in the digraph tables.
    // Values in the thirties indicate errors in the trigraph tables.
    const int get_status() const { return n_status; }

    const double get_a_score() const { return n_a_score; }
    const double get_c_score() const { return n_c_score; }

```

MAS.N 3-24-98 1:11p

Page 2 of 2

```

const double get_b_score() const { return n_b_score; }

void set_ag_threshold(const double in_threshold) { m_ag_threshold = in_threshold; }
void set_cg_threshold(const double in_threshold) { m_cg_threshold = in_threshold; }
void set_bg_threshold(const double in_threshold) { m_bg_threshold = in_threshold; }
void set_sg_threshold(const double in_threshold) { m_sg_threshold = in_threshold; }
void set_ag_threshold(const double in_threshold) { m_ag_threshold = in_threshold; }
void set_cg_threshold(const double in_threshold) { m_cg_threshold = in_threshold; }
void set_bg_threshold(const double in_threshold) { m_bg_threshold = in_threshold; }
void set_sg_threshold(const double in_threshold) { m_sg_threshold = in_threshold; }

const double get_ag_threshold() const { return m_ag_threshold; }
const double get_cg_threshold() const { return m_cg_threshold; }
const double get_bg_threshold() const { return m_bg_threshold; }
const double get_sg_threshold() const { return m_sg_threshold; }

};

#endif // _NLS_H
/* ..... */
/* ..... */

```

TDSLIB

```

// Copyright (C) 1998, Language Analysis Systems Inc.
//
// approx.cpp: implementation of the approx class.
//
//
// see my notes in the approx.h file for what I did to this
// class (Ara)
//
#include "stdafx.h"
#include <fstream>
#include "approx.h"
using namespace std;

ifdef _DEBUG
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#define new DEBUG_NEW
#undef THIS_FILE
#endif

// Construction/Destruction
//
//
// read a file of feature differences
const bool CApprox::set_rec_distances(const char *fname)
{
    ifstream f_in(fname);
    if (!f_in.is_open())
        return false;

    int i, j;
    char buf[BUFFSIZE]; // BUFFSIZE is apparently defined in <stdio.h>

    // first, set exact matches
    // there are 15 features and the file has the number of feature differences
    // multiplied by 10
    for (i = 0; i < 256; i++)
        m_fd_float_matrix[i][i] = (i == j ? 0.0 : 1.0);

    while(f_in.getline(buf, BUFFSIZE))
    {
        float n = atoi(buf + 4);
        if (*buf + 1) == BLANK && *buf + 3) == BLANK && n == 0)
        {
            x = buf(0);
            y = buf(2);
            m_fd_float_matrix[x][y] = n;
        }
    }
    f_in.close();
    return true;
}

// use this to read in a file containing rec codes and their
// associated distance scores
const bool CApprox::set_rec_distances(const char *filename)
{
    ifstream in_file(filename);
    if (!in_file.is_open())
        return false;

    int i, j;
    //char buf[BUFFSIZE]; // BUFFSIZE is apparently defined in <stdio.h>

    in_file.close();
    return true;
}

// here follow the code for the actual edit distance algo
// with several adaptations for testing purposes
//
// plain implementation of the edit distance algorithm
// this version does not use feature distance tables for
// anything.
//
const int CApprox::plain_edit_distance(const unsigned char *query,
                                       const unsigned char *variant,
                                       float &score)
{
    int rc = plain_diff(query, variant);
    score = get_plain_score();
    return rc;
}

float CApprox::get_plain_score()
{
    return 1.0 - m_diff / static_cast<float>(_max_in_str1_len, m_str2_len);
}

```


[illegible]

```

    {
        p_x = pi + 1;
        t_x = ti + 1;
        // Pick the lowest score from the rules
        // -- upper left
        lowest = m_rec_diff_array(p_x - 1)(t_x - 1) + (rec_eq(p_char, t_char, diff_matrix) ? 0
        : 1);
        // -- to the left
        d = m_rec_diff_array(p_x)(t_x - 1) + 1;
        if (d < lowest) lowest = d;
        // -- above
        d = m_rec_diff_array(p_x - 1)(t_x) + 1;
        if (d < lowest) lowest = d;
        // -- transposition
        if (pi > 1 && ti > 1)
        {
            d1 = rec_eq(arr1[pi - 1], arr2[ti], diff_matrix) ? 0 : 1;
            d2 = rec_eq(arr1[pi], arr2[ti - 1], diff_matrix) ? 0 : 1;
            if (d1 && 0 && d2 && 0)
            {
                d = m_rec_diff_array(p_x - 2)(t_x - 2) + d1 + d2 + 1;
                if (d < lowest)
                    lowest = d;
            }
            m_rec_diff_array(p_x)(t_x) = lowest;
        }
        m_str1_len = pi;
        m_str2_len = ti;
        m_rec_diff = get_rec_difference();
        return m_rec_diff;
    }
    // =====
    // REB implementation where the values are returned to
    // calculate fractional differences
    // this is the float version of the above function (yes I know about overloading)
    const float Opprox::rec_float_differences(const unsigned char *recArray1,
        const unsigned char *recArray2,
        float recCompArray[256][256],
        float score)
    {
        float rc = approx_rec_float_diff(recArray1, recArray2, recCompArray);
        score = get_rec_float_score();
        return rc;
    }
    const float Opprox::approx_rec_float_diff(const unsigned char *arr1,
        const unsigned char *arr2,
        float diff_matrix[256][256])
    {
        unsigned char p_char = NULL;
        unsigned char t_char = NULL;
        float d = 0.0; float d1 = 0.0; float d2 = 0.0;
        int pi = 0; int ti = 0; // string indexes
        int p_x = 0; int t_x = 0; // difference-array indexes
        float lowest = 0.0;
        for (pi = 0; (p_char = arr1[pi]) != EOS; pi++) // EOS is defined as '\0'
        {
            for (ti = 0; (t_char = arr2[ti]) != EOS; ti++)
            {
                p_x = pi + 1;
                t_x = ti + 1;
            }
        }
    }

```

```

    {
        p_x = pi + 1;
        t_x = ti + 1;
        // Pick the lowest score from the rules
        // -- upper left
        lowest = m_rec_diff_array(p_x - 1)(t_x - 1) + (rec_eq(p_char, t_char, diff_matrix) ? 0
        : 1);
        // -- to the left
        d = m_rec_diff_array(p_x)(t_x - 1) + 1;
        if (d < lowest) lowest = d;
        // -- above
        d = m_rec_diff_array(p_x - 1)(t_x) + 1;
        if (d < lowest) lowest = d;
        // -- transposition
        if (pi > 1 && ti > 1)
        {
            d1 = rec_eq(arr1[pi - 1], arr2[ti], diff_matrix) ? 0 : 1;
            d2 = rec_eq(arr1[pi], arr2[ti - 1], diff_matrix) ? 0 : 1;
            if (d1 && 0 && d2 && 0)
            {
                d = m_rec_diff_array(p_x - 2)(t_x - 2) + d1 + d2 + 1;
                if (d < lowest)
                    lowest = d;
            }
            m_rec_diff_array(p_x)(t_x) = lowest;
        }
        m_str1_len = pi;
        m_str2_len = ti;
        m_rec_diff = get_rec_difference();
        return m_rec_diff;
    }
    // =====
    const int Opprox::rec_differences(const unsigned char *recArray1,
        const unsigned char *recArray2,
        const unsigned char recCompArray[256][256],
        double score)
    {
        int rc = approx_rec_diff(recArray1, recArray2, recCompArray);
        score = get_rec_score();
        return rc;
    }
    // =====
    const int Opprox::approx_rec_diff(const unsigned char *arr1,
        const unsigned char *arr2,
        const unsigned char diff_matrix[256][256])
    {
        unsigned char p_char, t_char;
        int d1, d2;
        int pi, ti;
        int p_x, t_x;
        int lowest;
        for (pi = 0; (p_char = arr1[pi]) != EOS; pi++) // EOS is defined as '\0'
        {
            for (ti = 0; (t_char = arr2[ti]) != EOS; ti++)
            {
                p_x = pi + 1;
                t_x = ti + 1;
            }
        }
        // Pick the lowest score from the rules
        // -- upper left
        lowest = m_rec_diff_array(p_x - 1)(t_x - 1) + (rec_eq(p_char, t_char, diff_matrix) ? 0
    }

```



```

for (ti = 0; ti <= m_str2_len; ti++) {
    out_str.Format("%d ", m_diff_array[ti]);
    f_out << "LACSTR(out_str)";
}
f_out << endl;
}
f_out << endl;
/*
for (pi = 0; pi <= m_str1_len; pi++) {
    if (pi >= 1) f_out << "str1pl - ti << " "; else f_out << " ";
    for (ti = 0; ti <= m_str2_len; ti++) {
        out_str.Format("%d ", m_diff[pi][ti]);
        f_out << "LACSTR(out_str)";
    }
    f_out << endl;
}
f_out << endl;
*/
f_out.close();
#endif

```

0-1-4-2

```

copy (startPer, endPer, count);
count = count + 1;

```

```

retVal = consonantBitValArray(' ')[(unsigned char)*consonantPtr];
// next, loop through the pairs
while (*consonantPtr != '\0') {
    retVal |= consonantPairBitValArray[(unsigned char)*consonantPtr][((unsigned char)*consonantPtr
    * 2)];
    consonantPtr++;
}
// lastly, do the last character followed by a space
retVal |= consonantPairBitValArray[(unsigned char)(consonantPtr - 1)][(' ')];
return retVal;
}
*/
/*
unsigned int bitmapizeKV(const char *ucv)
{
    int    retVal = 0;
    const char *consonantPtr = ucv;
    while (*consonantPtr != '\0') {
        retVal |= consonantBitValArray[(unsigned char)*consonantPtr];
        consonantPtr++;
    }
    return retVal;
}
*/

```

```

consonantPtr1 = startPtr;
while (*consonantPtr1 != '\0') {
    consonantPtr2 = startPtr;
    while (*consonantPtr2 != '\0') {
        consonantPairBitValArray[(unsigned char)*consonantPtr1][(unsigned char)*consonantPtr2]
        .. int1 = bitValForInt1;
        consonantPairBitValArray[(unsigned char)*consonantPtr1][(unsigned char)*consonantPtr2]
        .. int2 = bitValForInt2;
        consonantPtr2++;
        // one or the other bitVal should be 0, since the entire 64 bit value should
        // have only 1 bit turned on.
        if (bitValForInt2 == 0) {
            bitValForInt1 += 1;
            // if we are at end of first 32 bits, set the second 32 bits to 1 and
            // start shifting that
            if (bitValForInt1 == 0)
                bitValForInt2 = 1;
        }
        else {
            bitValForInt2 += 1;
            // if we are at end of second 32 bits, set the first 32 bits to 1 and
            // start shifting that again
            if (bitValForInt2 == 0)
                bitValForInt1 = 1;
        }
    }
    consonantPtr1++;
}
delete [] startPtr;
*/

```

```

void bitmapizeKV(const char *ucv, unsigned int *bitSignature, int numBytesInSignature)
{
    // int    retVal = 0;
    const char *consonantPtr = ucv;

```

```

// first do the leading space with the first character.
bitSignature[0] = consonantPairBitValArray[' '][(unsigned char)*consonantPtr1].int1;
bitSignature[1] = consonantPairBitValArray[' '][(unsigned char)*consonantPtr1].int2;

```

```

// next, loop through the pairs
while (*consonantPtr != '\0') {
    bitSignature[0] |= consonantPairBitValArray[(unsigned char)*consonantPtr1][(unsigned char)*con
    .. consonantPtr + 1)].int1;
    bitSignature[1] |= consonantPairBitValArray[(unsigned char)*consonantPtr1][(unsigned char)*con
    .. consonantPtr + 1)].int2;
    consonantPtr++;
}

```

```

// lastly, do the last character followed by a space
bitSignature[0] |= consonantPairBitValArray[(unsigned char)*consonantPtr - 1]][(' ')].int1;
bitSignature[1] |= consonantPairBitValArray[(unsigned char)*consonantPtr - 1]][(' ')].int2;

```

```

// return retVal;
}

```

```

/*
unsigned int bitmapizeKV(const char *ucv)
{
    int    retVal = 0;
    const char *consonantPtr = ucv;
    // first do the leading space with the first character.

```

```

#define BE_DUP_MAX 255
#define FAST_MATCH 2048
#define INPUT_BUFFER_SIZE 1048576

#define todigit(c) ((c) - '0')

/*
The types symbol and superSymbol must both be unsigned, offset must be
signed type, state must be an unsigned type. A variable of type
superSymbol must be able to hold any valid value of symbol plus the
additional values defined following.
*/
typedef unsigned char symbol;
typedef unsigned superSymbol;
typedef long offset;
typedef unsigned state;

#define SYMBOLS 256
#define SYMBOL_START (SYMBOLS + 0)
#define SYMBOL_END (SYMBOLS + 1)
#define SYMBOL_START_LINE (SYMBOLS + 2)
#define SYMBOL_END_LINE (SYMBOLS + 3)
#define SYMBOL_START_SUPER (SYMBOLS + 4)
#define SYMBOL_END_SUPER (SYMBOLS + 5) /* Highest numbered valid superSymbol. */
#define SYMBOL_INVALID (SYMBOLS + 6)

#define FINAL ("isacat0")

typedef struct transitionStructure {
    struct transitionStructure *next;
    superSymbol sy;
    state sdd, scl;
} *transition;

typedef struct positionStructure {
    struct positionStructure *next;
    state sc;
    offset sc;
} *position;

typedef enum {
    MACHINE_FAST, MACHINE_SLOW, MACHINE_STOP, MACHINE_SUP_N, MACHINE_TEXT_LINE, MACHINE_X
} machineType;

typedef struct machineStructure {
    machineType type;
    transition (*trFastTable) [SYMBOL_SIZE];
    position *pFast;
    transition *trTable;
    transition trSigns, trStart, trEnd;
    state sddState;
    position ps;
    offset mapN, u, v, of;
} *machine;

typedef struct environmentStructure {
    struct environmentStructure *next;
    unsigned char *name, *body;
    int name;
} *environment;

typedef struct cytoesum {
    enum alpha, ascii, blank, cntrl, digit, graph, lower, print, punct, space,
    upper, xdigit, noneOfTheAbove
} cytoes;

```

```

static void * localMalloc (size_t n)
{
    void *ptr;

    if (n == 0)
        return (void *) 0;

    if ((ptr = malloc (n)) == (void *) 0)
    {
        AErrorMessage("Could not allocate memory!");
    }

    return ptr;
}

static transition
allocateTransition (void)
{
    transition tr;
    transition *trp;

    if (freeTransition == (transition *) 0)
        if (nranTransitionAllocations < MAX_TRANSITION_ALLOCATIONS)
        {
            static unsigned abunch = A_BUNCH;

            freeTransition = (struct transitionStructure *)MALLOC (abunch*sizeof (struct transitionStructure));

            for (i = 0, tr = freeTransition; i < (abunch - 1); i++, tr++)
                tr->next = tr + 1;

            tr->next = (transition *) 0;

            // save a pointer to what was just allocated so we can free it later
            nranTransitionAllocations[nranTransitionAllocations] = freeTransition;
            nranTransitionAllocations++;

            // next time, allocate twice as many
            abunch *= 1;
        }
        else
        {
            AErrorMessage("Too many transition allocations. Exiting");
            exit(0);
        }
    }

    tr = freeTransition;
    freeTransition = freeTransition->next;

    return tr;
}

#define NEW_TRANSITION(tr) (void) \
{ \
    freeTransition \
    ? ((tr) = freeTransition), freeTransition = freeTransition->next \
    : ((tr) = allocateTransition ()) \
}

#define DISPOSE_TRANSITION(tr) (void) \
{ \
    ((tr)->next = freeTransition, freeTransition = (tr)) \
}

static transition trStatic = (transition *) 0;
static transition savedTransition = (transition *) 0;

```

```

#define MALLOC(n) (localMalloc (n))
// #define REALLOC(ptr, n) (localRealloc ((ptr), (n)))
#define TRUE(p) (free (p))
// #define STRLEN(s) (strlen (localMalloc (strlen (s) + 1), (s)))

#define TRUE 1
#define FALSE 0

// how many times can we increase the pool of available transitions
#define MAX_TRANSITION_ALLOCATIONS 10
#define A_BUNCH 32

static int
static transition transitionAllocations = 0;
static transition freeTransition = (transition *) 0;
static int
static int
static char
static struct cMapStructure {
    char *c1;
    char *c2;
} cmap[] = {
    {"alpha", "alpha"}, {"ascii", "ascii"}, {"blank", "blank"},
    {"ctrl", "ctrl"}, {"digit", "digit"}, {"upper", "upper"}, {"lower", "lower"},
    {"print", "print"}, {"punct", "punct"}, {"space", "space"}, {"upper", "upper"},
    {"xdigit", "xdigit"}, {"char" 0, "nonorthodox"}
};

/* prototypes */
static transition eraseSet (
    unsigned char **expr, int compileFlag, environment env, int nparms,
    transition *parms
);

static transition tCopy (transition tr);
static transition tMerge (transition trA, transition trB);
static transition tConcatenation (
    transition trResult, transition trBranch, unsigned char op);
static transition tFree (transition tr);
static transition tEpsilon (transition trLeft, transition trRight);
static transition tIntersection (transition trLeft, transition trRight);
static transition tClosure (transition tr, unsigned closureType);
static transition tEpsilonBound (unsigned char **expr, int compileFlag, transition trAtom);
static transition tEpsilonBracket (
    unsigned char **expr, int compileFlag, environment env, int nparms,
    transition *parms
);

static transition tEpsilonSymbol (unsigned char **expr, int compileFlag);
static int tEpsilonBracket (unsigned char **expr, unsigned 'c1);
static transition tEpsilonSet (unsigned char **expr, int compileFlag, environment env);
static transition tCompact (transition tr);
static transition tReduce (transition tr);
static transition tEpsilon (transition tr);
static transition tConcatenated (transition tr, unsigned n);
static transition tClosure (transition tr, unsigned n);
static transition tEpsilonBracket (
    unsigned char **expr, int compileFlag, environment env, int nparms,
    transition *parms
);

static int tEpsilonBracketBracket (unsigned char **expr, unsigned match(), int tick);
static int tDigit (int c);
static environment tEpsilonSearch (environment env, char *name, int nparms);
static state tCompact_translate (state st, state *state, state *nextState);
static int tEpsilon (unsigned c, type cc);

```



```

static struct transitionstructure tCopyStructure;
static transition tCopy = tCopyStructure;

static transition
tConcatenate (transition tLeft, transition tRight)
{
    transition tr, tResult, trData = (transition) 0;
    int lambdaHead = 0;
    if (tLeft == tCopy || tRight == tCopy)
        return tCopy;
    if (tLeft == (transition) 0)
        if (tRight == (transition) 0)
            return (transition) 0;
        else
            return tRight;
    else if (tRight == (transition) 0)
        return tLeft;
    if (tLeft->sy == SYMBOL_LAMBDA)
    {
        tr = tLeft;
        tLeft = tLeft->next;
        DISPOSE_TRANSITION (tr);
        if (tRight->sy == SYMBOL_LAMBDA)
        {
            lambdaHead = 1;
            tr = tRight;
            tRight = tRight->next;
            DISPOSE_TRANSITION (tr);
        }
        trData = tCopy (tRight);
        for (tr = trData; tr = tr->next; )
        {
            if (tr->sed != 0)
                tr->sed == tLeft->sed + 1;
            if (tr->sel != FINISH)
                tr->sel == tLeft->sed + 1;
        }
    }
    if (lambdaHead)
        trData = tMerge (trData, tCopy (tLeft));
    else if (tRight->sy == SYMBOL_LAMBDA)
    {
        tr = tRight;
        tRight = tRight->next;
        DISPOSE_TRANSITION (tr);
        trData = tCopy (tLeft);
    }
    for (tr = tRight; tr = tr->next; )
    {
        tr->sed == tLeft->sed + 1;
        if (tr->sel != FINISH)
            tr->sel == tLeft->sed + 1;
    }
    for (tr = tLeft; tr = tr->next; )
        if (tr->sel == FINISH)
            tr->sel = tLeft->sed + 1;
}

```

```

tResult = tMerge (tLeft, tRight);
if (trData)
    tResult = tMerge (trData, tResult);
if (lambdaHead)
{
    NEW_TRANSITION (tr);
    tr->sy = SYMBOL_LAMBDA;
    tr->next = tResult;
    tResult = tr;
}
return tResult;
}

static transition
tCopy (transition tr)
{
    transition tr = (transition) 0, *trp = &tr;
    if (tr == tCopy)
        return tCopy;
    for (tr = tr->next;
        {
            transition tr0;
            NEW_TRANSITION (tr0);
            tr0->next = (transition) 0;
            tr0->sy = tr->sy;
            tr0->sed = tr->sed;
            tr0->sel = tr->sel;
            (*trp) = tr0;
            trp = &tr0->next;
        }
    return tr;
}

static transition tMerge (transition trA, transition trB)
{
    transition tr = (transition) 0, *trp = &tr;
    if (trA == tCopy || trB == tCopy)
        return tCopy;
    while (trA && trB)
    {
        int aFirst;
        transition tr0;
        if (trA->sed > trB->sed)
            aFirst = 1;
        else if (trA->sed < trB->sed)
            aFirst = 0;
        else if (trA->sel > trB->sel)
            aFirst = 1;
        else if (trA->sel < trB->sel)
            aFirst = 0;
        else if (trA->sy < trB->sy)
            aFirst = 1;
        else if (trA->sy > trB->sy)

```



```

    trAtom = erepConcatenate (expr, compileFlag, env, rname, name);
    if (trAtom == (transition) 0) { *expr != '\';
        ERE_PARSE_DIE (t);
    }
    break;

case '.':
    trBranch = trConcatenate (trBranch, trAtom);
    trAtom = erepSymbol (SYMBOL_START_LBR, compileFlag);
    break;

case '(':
    trBranch = trConcatenate (trBranch, trAtom);
    trAtom = erepSymbol (SYMBOL_END_LBR, compileFlag);
    break;

case '[':
    trBranch = trConcatenate (trBranch, trAtom);
    trAtom = erepSymbol (SYMBOL_START_LBR, compileFlag);
    break;

case '{':
    trBranch = trConcatenate (trBranch, trAtom);
    trAtom = erepSymbol (SYMBOL_START_LBR, compileFlag);
    break;

case '|':
    trBranch = trConcatenate (trBranch, trAtom);
    trAtom = erepSymbol (SYMBOL_END_LBR, compileFlag);
    break;

case '\\':
    trBranch = trConcatenate (trBranch, trAtom);
    trAtom = erepSymbol (SYMBOL_SIGMA, compileFlag);
    break;

case '\\':
    if (erepIsBackslash (expr, t))
        ERE_PARSE_DIE (t);
    else
    {
        trBranch = trConcatenate (trBranch, trAtom);
        trAtom = erepSymbol (t, compileFlag);
    }
    break;

case 'v':
    trBranch = trConcatenate (trBranch, trAtom);
    if (!trAtom = erepConcat (expr, compileFlag, env)) == (transition) 0)
        ERE_PARSE_DIE (t);
    break;

case 'g':
    {
        int pn;

        **expr;
        if (!isdigit (**expr))
            ERE_PARSE_DIE (t);
        pn = todigit (**expr);
        if (pn == 0 || pn > rname)
            return (transition) 0;
        trBranch = trConcatenate (trBranch, trAtom);
        if (compileFlag)
            trAtom = trCopy (rname[pn - 1]);
        else
            trAtom = trCopy;
    }
    break;

default:
    trBranch = trConcatenate (trBranch, trAtom);
    trAtom = erepSymbol (t, compileFlag);
    break;
}

return erepConcatenation (trResult, trConcatenate (trBranch, trAtom), 'op');
}

static transition trFree (transition tr)
{

```

```

    if (tr == trCopy)
        return (transition) 0;
    while (tr)
    {
        transition tr;
        tr = tr;
        tr = tr->next;
        DISPOSE_TRANSITION (tr);
    }
    return tr;
}

static transition trEthin (transition trLeft, transition trRight)
{
    transition tr, trResult;
    int lambdaNeeded = 0;

    if (trLeft == trCopy || trRight == trCopy)
        return trCopy;

    if (trLeft == (transition) 0)
        if (trRight == (transition) 0)
            return (transition) 0;
    else
        return trRight;
    else if (trRight == (transition) 0)
        return trLeft;
    else
    {
        if (trLeft->sy == SYMBOL_LAMBDA)
        {
            lambdaNeeded = 1;
            tr = trLeft;
            trRight = trRight->next;
            DISPOSE_TRANSITION (tr);
        }
        if (trRight->sy == SYMBOL_LAMBDA)
        {
            lambdaNeeded = 1;
            tr = trRight;
            trRight = trRight->next;
            DISPOSE_TRANSITION (tr);
        }
        for (tr = trRight; tr = tr->next;
            {
                if (tr->sy != 0)
                    tr->sy = trLeft->sy;
                if (tr->sy != 0 && tr->sy != trLeft->sy)
                    tr->sy = trLeft->sy;
            }

        trResult = trMerge (trLeft, trRight);
        if (lambdaNeeded)
        {
            NEW_TRANSITION (tr);
            tr->sy = SYMBOL_LAMBDA;
            tr->next = trResult;
            trResult = tr;
        }
    }
}

```

```

    return tResult;
}

static transition trInsect (transition trLeft, transition trRight)
{
    transition tr, tr0, tr1, tResult, *trp;
    int lambdaNeeded;
    state adjust;

    if (trLeft == trOkay || trRight == trOkay)
        return trOkay;

    if (trLeft == (transition) 0)
        if (trRight == (transition) 0)
            return (transition) 0;
        else
            return trRight;
    else if (trRight == (transition) 0)
        return trLeft;
    if (trLeft->sy == SYMBOL_LAMDA)
    {
        lambdaNeeded = 1;
        tr = trLeft;
        trLeft = trLeft->next;
        DISPOSE_TRANSITION (tr);
    }
    else
        lambdaNeeded = 0;

    if (trRight->sy == SYMBOL_LAMDA)
    {
        tr = trRight;
        trRight = trRight->next;
        DISPOSE_TRANSITION (tr);
    }
    else
        lambdaNeeded = 0;

    trp = trResult;
    adjust = trRight->sc0 + 1;
    for (tr0 = trLeft; tr0; tr0 = tr0->next)
        for (tr1 = trRight; tr1; tr1 = tr1->next)
            if (tr0->sy == tr1->sy || tr0->sy == SYMBOL_SIGMA || tr1->sy == SYMBOL_SIGMA)
            {
                state sc0 = tr0->sc0-adjust + tr1->sc0;
                state sc1;

                if (tr0->sc1 == FINAL)
                    if (tr1->sc1 == FINAL)
                        sc1 = FINAL;
                    else
                        continue;
                else
                    if (tr1->sc1 == FINAL)
                        continue;
                    else
                        sc1 = tr0->sc1-adjust + tr1->sc1;

                NEW_TRANSITION (tr);
                tr->sc0 = sc0;
            }
}

tr->sc1 = sc1;
tr->sy = tr0->sy == SYMBOL_SIGMA ? tr1->sy : tr0->sy;
(*trp) = tr;
trp = tr->next;
}

(*trp) = (transition) 0;
tResult = trCompact (trReduce (trCompact (trSort (tResult))));
if (tResult == (transition) 0)
{
    NEW_TRANSITION (tResult);
    tResult->next = (transition) 0;
    tResult->sc0 = 0;
    tResult->sc1 = FINAL;
    tResult->sy = SYMBOL_INVALID;
}
if (lambdaNeeded)
{
    NEW_TRANSITION (tr);
    tr->sy = SYMBOL_LAMDA;
    tr->next = tResult;
    tResult = tr;
}
return tResult;
}

static transition
// trClosure (transition trX, unsigned char closureType)
// trClosure (transition trX, unsigned closureType)
{
    transition tr, tr0, tResult;
    int lambdaNeeded;

    if (trX == trOkay || trX == (transition) 0)
        return trOkay;

    if (trX->sy == SYMBOL_LAMDA)
    {
        tr = trX;
        trX = trX->next;
        DISPOSE_TRANSITION (tr);
        lambdaNeeded = 1;
    }
    else if (closureType != '?')
        lambdaNeeded = 1;
    else
        lambdaNeeded = 0;

    if (closureType == '?')
        tResult = trX;
    else
    {
        transition trRestart = (transition) 0, *trpRestart = trRestart;
        for (tr = trX; tr; tr = tr->next)
            if (tr->sc1 == FINAL)
            {
                NEW_TRANSITION (tr0);
                tr0->next = (transition) 0;
                tr0->sy = tr->sy;
                tr0->sc0 = tr->sc0;
            }
    }
}

```

```

    tr0->rel = 0;
    tr->parent = tr0;
    tr->parent = &tr0->next;
}

tr->result = trMerge (tr, tr->parent);

if (!isEmpty())
{
    NEW_TRANSITION (tr);
    tr->xy = SYMBOL_LENGTH;
    tr->next = tr->result;
    tr->parent = tr;
}

return tr->result;
}

static transition trParseBounded (unsigned char **expr, int compileFlag, transition trAtom)
{
    unsigned lower = 0, upper = 0;
    unsigned c = **expr;
    transition trResult = (transition) 0;

    if (!isdigit (c))
    {
        trFree (trAtom);
        return (transition) 0;
    }

    if (c != '-') || upper < lower || upper == 0
    {
        trFree (trAtom);
        return (transition) 0;
    }

    if (compileFlag)
    {
        if (lower)
        {
            trResult = trConcatenate (trCopy (trAtom), lower);
            upper = lower;
        }

        if (upper)
            trResult = trConcatenate (trResult, trCloseup (trAtom, upper));
        else
            trFree (trAtom);
    }

    return trResult;
}
else
    return trOkay;
}

static transition trParseBracket (
    unsigned char **expr, int compileFlag, environment env, int rparms,
    transition *parms
)
{
    unsigned i;
    int match(SYMBOLS), matchStart, matchEnd, matchStartLine, matchEndLine;
    int tick = 1;

    if (**expr == '\0')
        return (transition) 0;

    if (**expr == '@')
    {
        return trParseBounded (&expr, compileFlag, env, rparms, parms);
    }

    if (**expr == '\n')
    {
        tick = 0;
        if (**expr == '\0')
            return (transition) 0;
    }

    for (i = 0; i < SYMBOLS; i++)
        match[i] = tick;
    matchStart = tick;
    matchEnd = tick;
    matchStartLine = tick;
    matchEndLine = tick;

    do
    {

```

```

    tr0->rel = 0;
    tr->parent = tr0;
    tr->parent = &tr0->next;
}

tr->result = trMerge (tr, tr->parent);

if (!isEmpty())
{
    NEW_TRANSITION (tr);
    tr->xy = SYMBOL_LENGTH;
    tr->next = tr->result;
    tr->parent = tr;
}

return tr->result;
}

static transition trParseBounded (unsigned char **expr, int compileFlag, transition trAtom)
{
    unsigned lower = 0, upper = 0;
    unsigned c = **expr;
    transition trResult = (transition) 0;

    if (!isdigit (c))
    {
        trFree (trAtom);
        return (transition) 0;
    }

    if (c != '-') || upper < lower || upper == 0
    {
        trFree (trAtom);
        return (transition) 0;
    }

    if (compileFlag)
    {
        if (lower)
        {
            trResult = trConcatenate (trCopy (trAtom), lower);
            upper = lower;
        }

        if (upper)
            trResult = trConcatenate (trResult, trCloseup (trAtom, upper));
        else
            trFree (trAtom);
    }

    return trResult;
}
else
    return trOkay;
}

static transition trParseBracket (
    unsigned char **expr, int compileFlag, environment env, int rparms,
    transition *parms
)
{
    unsigned i;
    int match(SYMBOLS), matchStart, matchEnd, matchStartLine, matchEndLine;
    int tick = 1;

    if (**expr == '\0')
        return (transition) 0;

    if (**expr == '@')
    {
        return trParseBounded (&expr, compileFlag, env, rparms, parms);
    }

    if (**expr == '\n')
    {
        tick = 0;
        if (**expr == '\0')
            return (transition) 0;
    }

    for (i = 0; i < SYMBOLS; i++)
        match[i] = tick;
    matchStart = tick;
    matchEnd = tick;
    matchStartLine = tick;
    matchEndLine = tick;

    do
    {

```



```

    trp = &tr;
    while (trX)
    {
        tr0 = trX;
        trX = trX->next;
        tr0->set0 = trCompact_translate(tr0->set0, &trTranslate, maxState);
        if (tr0->set0 == maxState + 1)
        {
            DISPOSE_TRANSITION(tr0);
            continue;
        }
        tr0->set1 = trCompact_translate(tr0->set1, &trTranslate, maxState);
        if (tr0->set1 == maxState + 1)
        {
            DISPOSE_TRANSITION(tr0);
            continue;
        }
        (*trp) = tr0;
        trp = &tr0->next;
    }
    (*trp) = (transition) 0;
    return tr;
}

transition trReduce(transition trX)
{
    state maxState;
    transition tr, tr0, tr1, *trp;
    transition *trSc;
    if (trX == trWay || trX == (transition) 0)
        return trX;
    maxState = trX->set0;
    trSc = (transition *) alloca(maxState + 1 * sizeof(transition));
    for (trp = trSc; trp <= trSc + maxState; trp++)
        *trp = (transition) 0;
    tr = (transition) 0;
    trp = &tr;
    while (trX)
    {
        tr0 = trX;
        trX = trX->next;
        if (tr0->set0 == 0)
        {
            tr0->next = (transition) 0;
            (*trp) = tr0;
            trp = &tr0->next;
        }
        else if (tr0->set0 <= maxState)
        {
            tr0->next = trSc[tr0->set0];
            trSc[tr0->set0] = tr0;
        }
        else
        {
            DISPOSE_TRANSITION(tr0);
        }
    }
    for (tr0 = tr; tr0; tr0 = tr0->next)
        if (tr0->set1 != FINAL)
            while (tr1 = trSc[tr0->set1])

```

```

    c = &c - todigit(*-->expr);
    if (!>expr[1] == 0) && >expr[1] <= '7')
        c = &c + todigit(*-->expr);
    if (c == SWSWGLS)
        return 1;
    break;
case 'x':
    if (todigit(*-->expr))
        c = todigit(*-->expr);
    else
        return 0;
    if (todigit(*-->expr[1]))
        c = &c + todigit(*-->expr);
    break;
}
*cX = c;
return 0;
}

static transition envParseSet(unsigned char *expr, int completeFlag, environment env)
{
    char name[2];
    unsigned char *body;
    name[0] = *-->expr;
    if (!isalpha(name[0]))
        return (transition) 0;
    name[1] = '\0';
    if (!env->envSwitch(env, name, 0) == (environment) 0)
        return (transition) 0;
    body = env->body;
    return envParse(body, 1, env->next, -1, (transition *) 0);
}

static transition trCompact(transition trX)
{
    transition tr, tr0, *trp;
    state maxState, prevState, *exp, *trTranslate;
    if (trX == trWay || trX == (transition) 0)
        return trX;
    maxState = 0;
    prevState = trX->set0;
    for (tr = trX->next; tr; tr = tr->next)
        if (tr->set0 != prevState)
        {
            maxState++;
            prevState = tr->set0;
        }
    trTranslate = (state *) alloca(maxState + 1 * sizeof(state));
    exp = &trTranslate;
    prevState = trX->set0 + 1;
    for (tr = trX; tr; tr = tr->next)
        if (tr->set0 != prevState)
            *exp++ = prevState = tr->set0;
}

```



```

tip = &trforward;
for (tr = trf; tr->next > 0; tr = tr->next)
{
    for (i = tr->N - 1; i > 0; --i)
    {
        REA_TRANSITION (tr0);
        tr0->next = 0;
        if (tr->next == FINAL)
            if (i == N - 1)
                tr0->next = FINAL;
            else
                tr0->next = (i + 1)*adjust;
        else
            tr0->next = tr->next + i*adjust;
        tr0->sy = tr->sy;
        *trp = tr0;
        trp = &tr0->next;
    }
    *trp = transition 0;
    tr = tr->next->next (tr, N);
    tr = &tr->next (tr, trforward);
    REA_TRANSITION (tr0);
    tr0->sy = SYMBOL_LENGTH;
    tr0->next = tr;
    tr = tr0;
}
return tr;
}

static transition reParseBracket {
    unsigned char **expr, int compileFlag, environment env, int upara,
    transition *para
}
{
    unsigned char *name, *body, *s, saved;
    int parmslew = 0;
    transition trResult, parmslew(9);
    name = **expr;
    if (!isalpha (**expr) && **expr != '_')
        return transition 0;
    do
    {
        **expr;
        while (!isalnum (**expr) || **expr == '_');
        s = *expr;
        while (**expr == ' ' || **expr == '\t')
            **expr;
        switch (**expr)
        {
            case '[':
                break;
            case '(':
                do
                {

```

CGREP.CPP 3-24-98 11:24a

```

if (parmslew == 9)
{
    while (parmslew)
        trFree (parmslew(-parmslew));
    return transition 0;
}
**expr;
parmslew(parmslew) =
    reParse (expr, compileFlag, env, upara, para);
if (parmslew(parmslew) == transition 0)
{
    while (parmslew)
        trFree (parmslew(-parmslew));
    return transition 0;
}
parmslew++;
while (**expr == ' ');
if (**expr != '\t')
{
    while (parmslew)
        trFree (parmslew(-parmslew));
    return transition 0;
}
**expr;
while (**expr == ' ' || **expr == '\t')
    if (**expr != '\t')
    {
        while (parmslew)
            trFree (parmslew(-parmslew));
        return transition 0;
    }
    break;
default:
    return transition 0;
}
saved = *s;
*s = '\0';
env = envSearch (env, (char *)name, parmslew);
*s = saved;
if (env == environment 0)
{
    while (parmslew)
        trFree (parmslew(-parmslew));
    return transition 0;
}
body = env->body;
trResult = reParse (body, compileFlag, env->next, parmslew, parmslew);
while (parmslew)
    trFree (parmslew(-parmslew));
return trResult;
}

static int reParseBracketHeader (unsigned char **expr, unsigned match[], int tick)
{
    if (**expr == ' ':)

```

Page 11 of 13

```

    unsigned char *e;
    struct ctmstructure *ctm;
    unsigned i;

    for (e = *expr + 1; *e != '\0'; e++)
        if (*e == '\0')
            return i;

    *e = '\0';
    for (ctm = ctmq; ctm->cte; ctm++)
        if (strcmp(ctm->cta, (const char *) *expr + 1) == 0)
            break;
    *e = '\0';

    if (
        ctm->cte == nondeterministic || e[1] != '-' || e[2] == '\0' || e[2] == '+'
    )
        return i;

    *expr = e + 2;

    for (i = 0; i < SYMBOLS; i++)
        if (istype(i, ctm->cte))
            match[i] = ctm;
    else
        match[i] = tick;

    return 0;
}

static transition trSymbol (superSymbol sy)
{
    transition tr;
    NON_TRANSITION (tr);
    tr->next = transition 0;
    tr->sy = sy;
    tr->act = 0;
    tr->rel = FINAL;

    return tr;
}

static int todigit (int c)
{
    if (isdigit(c))
        return (todigit(c));

    switch (c)
    {
        case 'a': case 'A':
            return 10;
        case 'b': case 'B':
            return 11;
        case 'c': case 'C':
            return 12;
        case 'd': case 'D':
            return 13;
        case 'e': case 'E':
            return 14;
        case 'f': case 'F':
            return 15;
    }
}

```

```

    return -1;
}

static environment envSearch (environment env, char *name, int options)
{
    for (; env; env = env->next)
        if (options == env->options && strcmp(name, (const char *) (env->name)) == 0)
            return env;
    return (environment) 0;
}

static state trCompile_translate (state st, state *stTranslate, state maxState)
{
    state *stp, *stp0, *stp1;
    return st;

    if (st > *stTranslate)
        return maxState + 1;

    stp0 = stTranslate;
    stp1 = stTranslate + maxState;
    while (stp0 <= stp1)
    {
        stp = stp0 + (stp1 - stp0) / 2;

        if (st > *stp)
            stp1 = stp - 1;
        else if (st < *stp)
            stp0 = stp + 1;
        else
            return (stTranslate + maxState) - stp;
    }

    return maxState + 1;
}

static int istype (unsigned c, ctype ct)
{
    switch (ct)
    {
        case alnum:
            return isalnum(c);
        case alpha:
            return isalpha(c);
        case ascii:
            return isascii(c);
        case blank:
            return ispace(c);
        case cntrl:
            return iscntrl(c);
        case digit:
            return isdigit(c);
        case graph:
            return isgraph(c);
        case lower:
            return islower(c);
        case print:
            return isprint(c);
        case punct:
            return ispunct(c);
    }
}

```

```

case space:
    return ispace (c);
case upper:
    return isupper (c);
case digit:
    return isdigit (c);
}

return 0;
}

int ParseExpr (unsigned char* expr)
{
    if (savedTransition) {
        tFree(savedTransition);
        savedTransition = NULL;
    }
    cgraph_environment env = CGRAPH_NULL_ENV;
    if (!trStatic = eParse (expr, 1, (environment) env, -1, (transition *) 0)) {
        savedTransition = trStatic;
        return TRUE;
    }
    else
        return FALSE;
}

/*
 * Function to dispose of all the transitions in the
 * static chain (which will place them back into the
 * available pool. We then free the available pool.
 */
void RunCleanup()
{
    if (savedTransition) {
        tFree(savedTransition);
        savedTransition = NULL;
    }
    for (int i = 0; i < numTransitionalAllocations; i++)
        free(TransitionalAllocations[i]);
}

int RunGetState (unsigned *se0, unsigned *se1, unsigned *cy)
{
    if (!trStatic)
    {
        *se0 = trStatic->se0;
        *se1 = trStatic->se1;
        *cy = trStatic->cy;
        trStatic = trStatic->next;
        return TRUE;
    }
    else
        return FALSE;
}

```

```
//
// include "stdafx.h"
// include "Distance.h"

int Distance[255][255];

void Distance::SetDistance(UCHAR c1, UCHAR c2, int dist)
{
    Distance[c1][c2] = dist;
    Distance[c2][c1] = dist;
}

void Distance::SetDefaults()
{
    a_SymbolThreshold = 60;
    for (UCHAR i=0; i<255; i++)
    {
        for (UCHAR j=0; j<255; j++)
        {
            if (i==j)
            {
                Distance[i][j] = Distance[j][i] = 0;
            }
            else
            {
                Distance[i][j] = Distance[j][i] = 100;
            }
        }
    }
}

// Distance::SetDistance(LPCSTR line)
{
    // Set distance from a string of format
    // A B VALUE
    if (strlen(line, "Threshold", strlen("Threshold")) == 0)
    {
        a_SymbolThreshold = atoi( strlen("Threshold") );
        return TRUE;
    }
    CString str = line;
    char *token;
    char seps[] = " ,\t ";
    UCHAR a,b;
    int Value;
    int icon = 0;
    token = strtok( str.GetBuffer(BUFSIZ), seps );
    while( token != NULL )
    {
        switch (item)
        {
            case 0:
                a = *token;
                break;
            case 1:
                b = *token;
                break;
            case 2:
                Value = atoi(token);
                break;
            default:
                break;
        }
    }
}
```

```
break;
}
token = strtok( NULL, seps );
item++;
}
if (item == 2)
    SetDistance(a,b,Value);
return FALSE;
}

// Distance::Reader(LPCSTR Infile /*= NULL*/)
{
    BOOL Read = TRUE;
    if (Infile==NULL)
        Infile = "Distance nul";
    if (!err.IsOpen())
        err.open("Error.Log", ios::out|ios::trunc);
    if (!err.IsOpen())
        ErrorMessage("Could not open error log file: Error.Log");
    err << "Reading Distance Information " << Infile << endl;
    CString strLine, strItem;
    CString left,match,right,output;
    SetDefaults();
    ifstream in(Infile, ios::in);
    if (!in || in.bad())
    {
        err << "Problem with input file " << Infile << endl;
        err.close();
        return FALSE;
    }
    while( !in.getline(strLine.GetBuffer(BUFSIZ),BUFSIZ) )
    {
        strLine.ReleaseBuffer();
        strLine.Truncate();
        strLine.Truncate();
        if (strLine.IsEppc())
        {
            if (strLine[0] == ':')
                continue;
            SetDistance(strLine);
        }
    }
    // when to close the err file hmm.. let's leave it open so we
    // can include error logs while the names are being translated
    // err.close();
    return Read;
}
```

```

Copyright (C) 1998, Language Analysis Systems Inc.

//
// euclidistance.cpp: Implementation of the CObidDistance class.
//
#include "stdafx.h"
#include "obidist.h"

//=====
// Construction/Destruction
//=====

CObidDistance::CObidDistance(CSimpleCodeDictionary *aSimpleCodeDictionary)
{
    //Initialize distance array to all 0's
    int i = 0;
    while (i < STRINDEXMAX-1) {
        for (j = 0; j < (STRINDEXMAX-1); j++)
            distanceArray[i][j] = 0;
        i++;
    }
    threshold = 0;
    thresholdInt = 0;
    simpleCodeDictionary = aSimpleCodeDictionary;
}

CObidDistance::~CObidDistance()
{
}

//=====
// Member Functions
//=====

int CObidDistance::getDistanceArrayCall(int i, int j)
{
    if ((i > STRINDEXMAX-1) || (j > STRINDEXMAX-1))
        return(-1);
    else
        return(distanceArray[i][j]);
}

float CObidDistance::getDistanceScore(unsigned char *str1)
{
    int i;
    int j;
    int size1;
    int size2;
    unsigned char *scr1;
    unsigned char *scr2;
    unsigned char *scr3;
    int diff;
    int lowest;
    int maxize;
    float distance;

    //get size of scr1
    size1 = strlen((char*)scr1);

```

```

    str1 = str1Start;
    while (*str1 != EOS)
    {
        distanceArray[0][i+1] = simpCodeDistance->get_simpcode_distance(0,*str1) +
            distanceArray[0][i];
        str1++;
    }

    //compute rest of array starting with second row
    i = 1;
    str1 = str1Start;
    while (*str1 != EOS)
    {
        curDiff = simpCodeDistance->get_simpcode_distance(0,*str1) +
            distanceArray[i-1][0];
        distanceArray[i][0] = curDiff;

        j = 1;
        str1 = str1Start;
        while (*str1 != EOS)
        {
            //coming from the left
            lowest = simpCodeDistance->get_simpcode_distance(str1,0) +
                distanceArray[i][j-1];
            diff = simpCodeDistance->get_simpcode_distance(0,*str1) +
                distanceArray[i][j-1];
            if (diff < lowest)
                lowest = diff;
            //coming from above
            diff = simpCodeDistance->get_simpcode_distance(0,*str1) +
                distanceArray[i-1][j];
            if (diff < lowest)
                lowest = diff;
            diff = simpCodeDistance->get_simpcode_distance(str1,0) +
                distanceArray[i-1][j];
            if (diff < lowest)
                lowest = diff;
            //coming from above left
            diff = simpCodeDistance->get_simpcode_distance(str1,*str1) +
                distanceArray[i-1][j-1];
            if (diff < lowest)
                lowest = diff;
            distanceArray[i][j] = lowest;
            if (lowest < curDiff)
                curDiff = lowest;
            j++;
            str1++;
        }
        if (curDiff > maxDiff)
            return(0,0);
        str1++;
    }
    distance = distanceArray[i-1][j-1];
    return(1.0 - ((distance/DIFFMAX)/maxSize));
}

```

This function is identical to getDistanceScore, except that it includes code to abort the routine once the number of matches a pre-specified threshold. This threshold is specified via the setThreshold() method. Note that this method rearranges the order of the strings being compared such that the cost is calculated as the larger string being transcribed to the smaller string.

unsigned char *str2)

```

{
    int i;
    int j;
    int size1;
    int size2;
    unsigned char *str1;
    unsigned char *str2;
    unsigned char *str1Start;
    unsigned char *str2Start;
    int lowest;
    int diff;
    int maxDiff;
    int curDiff;
    int maxSize;
    float distance;

    //get size of str1
    size1 = strlen((char*)str1);
    if (size1 > STRLENMAX)
        return(-1);

    //get size of str2
    size2 = strlen((char*)str2);
    if (size2 > STRLENMAX)
        return(-2);

    // set longer string to str1 and set shortest to str2
    // also note the max size
    if (size2 > size1)
    {
        // second string is bigger
        maxSize = size2;
        str1Start = str2;
        str2Start = str1;
    }
    else
    {
        // first string is bigger
        maxSize = size1;
        str1Start = str1;
        str2Start = str2;
    }

    //calculate maximum number of differences to stay below the threshold
    maxDiff = ((DIFFMAX - thresholdInc) * maxSize);

    //first cell is 0
    distanceArray[0][0] = 0;

    //Initialize first row - left only
    j = 0;
}

```

```
//
// .....
//
```

ExcName.cpp: implementation of the excName class.

See ExcName.h for documentation.

```
.....
//
```

```
#include "excName.h"
#include "ExcName.h"
```

```
#ifdef _DEBUG
#ifdef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
#endif
#endif
```

```
.....
//
```

```
.....
//
```

```
.....
//
```

```
ExcName::ExcName()
{
    nameCode[0] = '\0';
}
```

ExcName::ExcName(const string &n, const float p, const byte c),

== at 'valueCode'.

const char v, const bool e, ch
e_id, culture aClassCulture,
e_id, culture aPepCulture)

```
{
    ExcName(); // set up data specific to this subclass
    string(nameCode, nameCode, TDS_MAX_NAME_CODE);
    nameCode(TDS_MAX_NAME_CODE) = '\0';
}
```

void ExcName::assign(const ExcName &c)

```
{
    Name::assign(c);
    strcpy(nameCode, c.nameCode);
}
```

```
.....
//
```

// feature distance object implementation file

The CFeatureDistanceTable class encapsulates the feature distance table. The feature distance table is composed of the IPA characters and their feature distances stored as float values.

m_feature_distance_table contains the values m_status is a generic status variable that can be checked with the get_status() function. If m_status is false then there is a problem with the table. Usually this will be due to the dist.dat file not being loaded in properly. the overloaded function get_feature_distance can take either two ints or two unsigned chars as the arguments and will return a float value that is the feature distance between those two IPA characters.

```
#include "stdafx.h"
#include "CFeatureDistanceTable.h"
// #include "defines.h"
// #include "resource.h"

CFeatureDistanceTable::CFeatureDistanceTable(const char *name)
```

```
{
    int i,j;

    m_status = true; // default

    // initialize the array
    // 1 = maximum difference
    // 0 = no difference
    for(i = 0; i < 256; i++)
        m_feature_distance_table[i][i] = (float)(i == j ? 0.0 : 1.0);

    m_status = load_array(name);
    //amp();
}
```

```
CFeatureDistanceTable::~CFeatureDistanceTable()
{
}
```

// read the feature distance file and load the specified positions in the array.

// the format of the file should be exactly like this

// c c n.mmm...

//

// c = ipa characters

// n.mmm is some float value

// the ipa characters should be in positions 0 and 2

// the float begin at position 5.

// remember this is C++ and arrays start from 0

bool CFeatureDistanceTable::load_array(const char *name)

```
{
    FILE *source;
    unsigned char buf[255];
    char msg[255];
    unsigned int x,y;
```

if((source = fopen(name,"r")) == NULL)

```
{
    sprintf(msg,"Could not open %s for reading/make sure it exists.",name);
    AMessageBox(msg);
    return false;
}

// buf needs to be cast to char* for fgets to work
while(fgets(char *buf,255,source))
{
    float n = (float)atof((const char*)buf + 4);
    x = buf[0];
    y = buf[2];
    m_feature_distance_table[x][y] = n;
}

fclose(source);
return true;
}

// amp the table to a text file
void CFeatureDistanceTable::amp()
{
    int i,j;
    FILE *fdamp;
    char msg[255];

    if((fdamp = fopen("fdiscamp.txt","w")) == NULL)
    {
        sprintf(msg,"Could not open fdiscamp.txt for writing");
        AMessageBox(msg);
        return;
    }

    for(i = 0; i < 256; i++)
    {
        for(j = 0; j < 256; j++)
        {
            sprintf(fdamp,"%1.1f ",m_feature_distance_table[i][j]);
        }
        sprintf(fdamp,"\n");
        fclose(fdamp);
    }
}
```



```

// Copyright (C) 1998, Language Analysis Systems Inc.
//
// FeatureBidiDistance.cpp: Implementation of the FeatureBidiDistance class.
//
// =====
//
// #include "FeatureBidiDistance.h"
//
// =====
//
// Construction/Destruction
//
// =====
//
// FeatureBidiDistance::FeatureBidiDistance(CFeatureBidiDistanceTable *aFeatureBidiDistanceTable)
// {
//     // Initialize distance array to all 0's
//     int i = 0;
//     for (i = 0; i < (STRINFMAX+1); i++)
//     {
//         for (j = 0; j < (STRINFMAX+1); j++)
//             distancearray[i][j] = 0;
//     }
//     threshold = 0;
//
//     featureBidiTable = *aFeatureBidiTable;
//
//     FeatureBidiDistance::FeatureBidiDistance(i)
// {
// }
//
// Member functions
//
// =====
//
// float FeatureBidiDistance::getDistanceScore(unsigned char *str1,
//                                         unsigned char *str2)
// ..
// {
//     int i;
//     int j;
//     int size1;
//     int size2;
//     int *scr1;
//     unsigned char *scrifirst;
//     unsigned char *scri;
//     unsigned char *scrifirst;
//     unsigned char *scrilast;
//     float diff;
//     float lowest;
//     int midsize;
//     float distance;
//
//     // get size of str1
//     size1 = strlen((char*)str1);
//     if (size1 > STRINFMAX)
//         return(-1);
//
//     // get size of str2
//     size2 = strlen((char*)str2);

```

```

// includes code to abort the routine once the number of reaches a
// pre-specified threshold. This threshold is specified via the
// setThreshold() method.
// Note that this method re-arranges the order of the strings being
// compared such that the cost is calculated as the longer string
// being transformed to the shorter.
float FeatureDistance::getDistanceScoreWithMemory(unsigned char *str1,
{
    int i;
    int j;
    int size1;
    int size2;
    unsigned char *str1;
    unsigned char *str2;
    unsigned char *str1Start;
    unsigned char *str2Start;
    float lowest;
    float maxDiff;
    int maxSize;
    float distance;

    //get size of str1
    size1 = strlen((char*)str1);
    if (size1 > STRLENMAX)
        return(-1);

    //get size of str2
    size2 = strlen((char*)str2);
    if (size2 > STRLENMAX)
        return(-2);

    // set longer string to str1 and set shortest to str2
    // also note the max size
    if (size2 > size1) {
        // second string is bigger
        maxSize = size2;
        str1Start = str2;
        str2Start = str1;
    }
    else {
        // first string is bigger
        maxSize = size1;
        str1Start = str1;
        str2Start = str2;
    }

    //calculate maximum number of differences to stay below the threshold
    maxDiff = (1 - threshold) * maxSize;

    //first cell is 0
    distanceArray[0][0] = 0;

    //initialize first row - left only
    j = 0;
    str1 = str1Start;
    while (*str1 != EOS)
    {
        distanceArray[0][j+1] = featureDistance->get_feature_distance(0,*str1) +
            distanceArray[0][j];
        str1++;
        j++;
    }
}

```

FEATURE7.CPP 3-24-98 11:24a

```

//compute rest of array starting with second row
i = 1;
str1 = str1Start;
while (*str1 != EOS)
{
    curDiff = featureDistance->get_feature_distance(0,*str1) +
        distanceArray[i-1][0];
    distanceArray[i][0] = curDiff;

    j = 1;
    str1 = str1Start;
    while (*str1 != EOS)
    {
        //coming from the left
        lowest = featureDistance->get_feature_distance(*str1,0) +
            distanceArray[i][j-1];
        diff = featureDistance->get_feature_distance(0,*str1) +
            distanceArray[i-1][j-1];
        if (diff < lowest)
            lowest = diff;
        //coming from above
        diff = featureDistance->get_feature_distance(0,*str1) +
            distanceArray[i-1][j];
        if (diff < lowest)
            lowest = diff;
        diff = featureDistance->get_feature_distance(*str1,0) +
            distanceArray[i-1][j];
        if (diff < lowest)
            lowest = diff;
        //coming from above left
        diff = featureDistance->get_feature_distance(*str1,*str1) +
            distanceArray[i-1][j-1];
        if (diff < lowest)
            lowest = diff;
        distanceArray[i][j] = lowest;
        if (lowest < curDiff)
            curDiff = lowest;
        j++;
        str1++;
    }
    if (curDiff > maxDiff)
        return(0.0);
    i++;
    str1++;
}
distance = distanceArray[i-1][j-1];
return (float)(1.0 - (distance/maxSize));
}

```

Page 2 of 2

```
// Copyright (C) 1998, Language Analysis Systems Inc.
//
#include <sys/types.h>
#include <sys/stat.h>
#include <iostream>
#include <string>
using namespace std;

#include "stdafx.h"
#include "UCMap.h"
#include "tde_util.h"

// see the note about the original location of this function
// in the UCMap.cpp file.

bool UCMap::getMapEntryForUCV(const char *ucv_map_entry_5 *returnMapEntry)
{
    bool found = FALSE;
    bool done = FALSE;
    int top = 0;
    int bottom = mapEntries - 1;
    unsigned int testPosition;
    unsigned int jumpOne = 0;
    int completeCode;
    int bytestream;
    int tempString;

    while ((done) {
        testPosition = top + ((bottom - top) / 2);
        if (!seek(ucvMapFile, testPosition * mapEntrySize, SEEK_SET == 0)) {
            bytestream = fread((char *)returnMapEntry, 1, mapEntrySize, ucvMapFile);
            if (bytestream != mapEntrySize) {
                CSString msg;
                msg.Format("Error: only read %d bytes from UCV Map file. Wanted %d",
                    bystream, mapEntrySize);
                AfxMessageBox(msg);
            }
        }
        if (completeCode == 0) {
            tempString = ((char *)returnMapEntry) + ucwOffsetInStructure;
            completeCode = tde_unsigned_atocmp((const unsigned char *)tempString,
                (const unsigned char *)ucw));
        }
        if (completeCode == 0) {
            found = TRUE;
            done = TRUE;
        }
        else {
            if (top == bottom)
                done = TRUE;
            else {
                if (completeCode < 0) {
                    // map value is less than the one we are looking for, so 1
                    // into the map. That is, make the top the test position
                    if (top == testPosition)
                        top ++;
                    else
                        top = testPosition;
                }
            }
        }
    })
}
```

```

//
// function to generate the consonant variants for a name
// using the rule set.
//
// I placed this code in a separate file so that
// I could use optimization, which causes a problem with the rest
// of parse.cpp, and
// so that I could compile this more quickly.

```

```

#include "scdex.h"
#include "parse.h"
#include "constindex.h"
#endif

```

```
extern ofstream err;
```

```

//
// generates the (unique) variants for a name.
// The name can be a regular expression (in which case translateName
// should be FALSE), or just a string that needs to have rules applied
// to it (in which case translateName should be TRUE). If we are
// translating, the name will automatically be surrounded by spaces
// if it is not already.
// The truncate parameter can be used to truncate values. Any duplicates
// caused by such truncation are discarded. A truncate of 0 indicates
// that no truncation should occur.
// The devoiced flag, when TRUE, indicates that vowels should be removed
// from the generated variants. Again, any duplicates
// caused by such modification are discarded.
// vector<CString> = Ruleset::generate(LASTN name, BOOL devoiced,

```

```

//
//      BOOL translateName, int truncate,

```

```

//
//      tds_vargen_code arc)
//
//      vector<CString> *variants = new vector<CString>;
//      char
//      char
//      CString
//      CString
//      CString
//      CString
//      int

```

```

//
//      // its updated when devoicing

```

```

rc = TDS_VARGEN_CODE_OK;
if (translateName) {
    if (name != "") {
        strcpy(spacedName, "");
        strcpy(spacedName + 1, name);
        strcpy(spacedName);
        strcpy(spacedName, "");
        namePtr = spacedName;
    }
    else {
        namePtr = (char *)name;
    }
}

```

```

//
//      // name is already a regular expression
//
//      namePtr = "";
//      rc = name;

```

GENVAR.cpp 3-24-98 11:24a

```

//
// (new Isempty() == FALSE)
// If (ndata.get(new) {
//
//      const char
//      const
//      CString
//      BOOL
//
//      err;
//      IsDup;
//
//      ndata.get(0, err);
//      ndata.getForThisName = 0;
//
//      // go through and add the variants to our local list for this
//      // name. If requested, deap and remove vowels.
//      for (int i = 0; i < ndata.m_variants.GetSize(); i++) {
//          variantString = ndata.m_variants[i];
//
//          // remove vowels
//          if (devoiced) {
//              variantString = variantString.Devoice(1000);
//              for (p = q = variantStringPtr; p != '\0'; p++)
//                  if (search(AMLO_CONSONANTS, p) != NULL)
//                      *q++ = *p;
//              *q = '\0';
//
//          // truncate the UCV string to the size specified by
//          // ucv_len
//          if (truncate > 0) {
//              if ((p = variantStringPtr) > truncate)
//                  variantStringPtr(truncate) = '\0';
//          }
//
//          variantString.ReleaseBuffer();
//
//          // now deap
//          IsDup = FALSE;
//          for (int j = 0; j < ndata.m_forThisName; j++)
//              if (variantString == (*variants)[j]) {
//                  IsDup = TRUE;
//                  break;
//              }
//
//          if (IsDup == FALSE) {
//              variants->push_back(variantString);
//              ndata.m_forThisName++;
//          }
//          else {
//              variants->push_back(variantString);
//          }
//      }
//      else {
//          char
//          err[1000 + 1];
//          sprintf(err, "Error in regular expression: %s\n", rex, name);
//          AddressBox(err);
//          err << "Error in regular expression: " << (LASTN) rex << endl;
//          err << "name was: " << (LASTN) namePtr << endl;
//          rc = TDS_VARGEN_CODE_BAD_REGEX;
//      }
//
//      // redef;
//
//      //

```

return values:

}

GENVAR-1.CPP: 3-24-98 11:24a

Page 2 of 2

GROUPD-1.CPP 3-24-98 11:24a


```

1.
if (usingMemory)
{
    arrayIndex = startOffset / sizeof(unsigned int);
    if ((arrayIndex * numOffsets) > totalOffsets)
    {
        CString msg;
        msg.Format("Error: File %s, request for invalid block of 1d name offset vectors "
        "start
        (LACTS
        numOff

        .. ing at offset 1d in Group Names Offsets file".
        .. TR groupNamesOffsetsFileNam.
        .. succ, startOffset);
        AfxMessageBox(msg);
        rc = FALSE;
    }
    else
    {
        unsigned int *offsetPtr = &groupNamesOffsets[arrayIndex];
        for (i = 0; i < numOffsets; i++)
        {
            offsetSet->insert(*offsetPtr);
            offsetPtr++;
        }
    }
}
else
{
    // file based
    unsigned int tempNameOffset;
    // make sure the reads that follow will not attempt to read past the
    // end of the file
    if ((startOffset + (numOffsets * sizeof(unsigned int))) > lastOffset)
    {
        rc = false;
        CString msg;
        msg.Format("Error: File %s, request for invalid block of 1d name offset vectors "
        "start
        (LACTS
        numOff

        .. ing at offset 1d in Group Names Offsets file".
        .. TR groupNamesOffsetsFileNam.
        .. succ, startOffset);
        AfxMessageBox(msg);
    }
    else
    {
        if (seek(groupNamesOffsetsFile, startOffset, SEEK_SET) == 0)
        {
            for (i = 0; i < numOffsets; i++)
            {
                fread(&tempNameOffset, sizeof(unsigned int), 1, groupNamesOffsetsFile);
                offsetSet->insert(tempNameOffset);
            }
        }
        else
        {
            CString msg;
            msg.Format("Error: File %s, could not seek to offset 1d in Group Names "
            "
            .. "Offsets file ".
            .. (LACTS) groupNamesOffsetsFileNam.
            .. startOffset);
            AfxMessageBox(msg);
            return rc;
        }
    }
}

```



```

//
// BOOL QITA::leftMatch(LPCTSTR name, int len, int *pCount)
//
{
    ASSERT(!nameIsNull());

    if (!name) return TRUE;

    if (! strcmp( m_LastChars, name[len-1] ))
    {
        *pCount = 0;
        return FALSE;
    }

    CString name(name.len());
    name.MakeLowercase();
    return LeftStringMatch(FINAL, name, pCount)
}

// LeftStringMatch is like the other Walk functions
// it works on reversed QITA objects. Other than special control
// FURLs, and state 0 its much the same as the others which
// set to the number of characters which matched.
//
// BOOL QITA::LeftStringMatchWalk(unsigned int state, LPCTSTR lpName)
//
{
    ASSERT(!m_NameIsNull());

    BOOL bRet = FALSE;

    int state_index = ( state==FINAL ? 0 : state );

    BOOL bMatched = FALSE;
    for ( int i=0; i<m_states[state_index].GetSize(); i++)
    {
        unsigned ThisChar = m_states[state_index]
            [ i ];

        if ( *name == ThisChar || ThisChar == SYMBOL )
        {
            if (bMatched)
            {
                bMatched = TRUE;
                (*pCount)++;
            }

            if ( ! ( i==m_states[state_index].GetSize() ) )
            {
                // Matched the whole regular expression
                // just yet, there may be more
                bRet = TRUE;
            }

            if ( (*name-1) == '\0' )
            {
                return TRUE;
            }
            else
            {
                // this path is
                // so go no further
                if ( LeftStringMatchWalk(
                    state_index+1, pCount ) )
                {
                    // more path to go
                }
            }
            else if ( (*name-1) == '\0' )
            {
                // so go no further
                if ( LeftStringMatchWalk(
                    state_index+1, pCount ) )
                {
                    // more path to go
                }
            }
        }
    }
}

```

```

    }
    } // endif name==thisChar
    }
    return best; // didn't get out of this state
}

// The following two functions are like SubMatch() and SubStringMatchWalk()
// are like the corresponding submatch function except they will only return
// TRUE if the first state is found in both the CHA and the string.
//
// BOOL CHA::CompleteMatch(LPCSTR name, int *pCount)
// {
//     if (m_IsSigna) return TRUE;
//     if (! strchr( m_FirstChars, *name ) )
//     {
//         *pCount = 0;
//         return FALSE;
//     }
//     return CompleteStringMatchWalk(0, name, pCount);
// }
//
// walks until both final states are found or returns FALSE
//
// BOOL CHA::CompleteStringMatch(TCHAR state, LPCSTR name, int *pCount)
// {
//     BOOL bCharMatched = FALSE;
//     BOOL bRet = FALSE;
//     for ( int i=0; i<m_states[state].GetSize(); i++)
//     {
//         unsigned ThisChar = m_states[state].m_aTrans[i].symbol;
//         if ( *name == ThisChar || (unsigned)ThisChar == (unsigned) (TCHAR)SYMBOL_STORE )
//         {
//             if (!bCharMatched)
//             {
//                 bCharMatched = TRUE;
//                 (*pCount)++;
//             }
//             if ( (int) (m_states[state].m_aTrans[i].next) == -1 ) // found FINAL
//             {
//                 // Matched the whole regex so return if we also found
//                 // the end of the string
//                 if ( *name-i == '\0' ) // found BOS
//                 {
//                     return TRUE;
//                 }
//                 else
//                 {
//                     // this path is done and haven't reached BOS
//                     // so go no further down this path
//                 }
//             }
//             else // more path to go
//             {
//                 if ( *name-i == '\0' ) // no more name to go
//                 {
//                     // so go no further
//                 }
//                 else
//                 {
//                     if ( CompleteStringMatchWalk( m_states[state].m_aTrans[i].next, name+i,
//                         *pCount ) )
//                     {
//                         return TRUE;
//                     }
//                 }
//             }
//         }
//     }
// }

```

```

    if ( !inc) mfa_m_states[thatState].m_atrans[that_i].next == -1 ) // f
    {
        return TRUE;
    }
    else
    {
        // This path is done and haven't reached the end of that path
        // so go no further down that path
    }
}
else // more this path to go
{
    if ( !inc) mfa_m_states[thatState].m_atrans[that_i].next == -1 ) // f
    {
        // so go no further
    }
    else
    {
        if ( !NewMatchWalk( nfa,
            mfa_m_states[thatState].m_atrans[that_i].next,
            nfa_m_states[thatState].m_atrans[that_i].next, level+1
        ))
        {
            return TRUE;
        }
    }
}
} // endif ThisChar==ThatChar
}
return FALSE; // didn't get out of this state
}

// This Method Set() takes a regular expression and
// calls the parsing routines to retrieve the transition
// information. It calls the Add() method to create the state table;
// The removeall() method cleans up the CHA so that fresh info can be
// stored. Note the parameter Reverse which causes the states to be added
// in reverse order. This is for regular expressions on the left-hand side
// of a rule which have to scan in reverse order.
//
// BOOL CHA::Set(LPCSTR plex, BOOL bReverse )
// {
//     Removeall();
//     m_bMatcha = FALSE;
//     m_bSigna = FALSE;
//     m_bBoundry = FALSE;
//     // If regex is signa nothing else really matters
//     if (strcmp(plex, "-")==0)
//         m_bSigna = TRUE;
//     if (strcmp(plex, "*")==0)
//         m_bBoundry = TRUE;
//     unsigned sh=0, ei=0, sy=0;
//     char buf[BUFSIZ+1];
//     strcpy(buf, plex, BUFSIZ);
//     m_bReverse = bReverse;
//     Removeall();
//     m_FirstChars.Empty();
//     m_LastChars.Empty();
//     if ( !RegexParse(unsigned char*)buf )
//         return FALSE;
// }

```

Page 4 of 7

```

}
if ( !inc) mfa_m_states[thatState].m_atrans[that_i].next == -1 ) // found FINAL
{
    // Matched the whole regex so return true, but not
    // just yet, there may be more of the string to match
    bRet = TRUE;
}
if ( *(name+1) == '\0' ) // found EOS
{
    return TRUE;
}
else
{
    // this path is done and haven't reached EOS
    // so go no further down this path
}
else // more path to go
{
    if ( *(name+1) == '\0' ) // no more name to go
    {
        // so go no further
    }
    else
    {
        if ( StringMatchWalk( mfa_m_states[thatState].m_atrans[that_i].next, name+1, pCount
        ))
        {
            return TRUE;
        }
    }
} // endif name==ThatChar
}
return bRet; // didn't get out of this state
}

// The following two methods are like the methods above except that instead
// of comparing the CHA against a string they compare with another CHA
// Instead of just moving on to the next character in a string as above, these
// function move to the next state in m_states[]
//
// BOOL CHA::Match(CHAS nfa)
// {
//     if ( m_FirstChars.FindMatch(nfa.m_FirstChars) == -1 )
//         return FALSE;
//     return NewMatchWalk(nfa, 0, 0);
// }

// BOOL CHA::NewMatchWalk(CHAS nfa, TCHAR ThisState, TCHAR ThatState, int level)
// {
//     unsigned ThisChar;
//     for ( int This_i=0; This_i<m_states[ThisState].GetSize(); This_i++ )
//     {
//         ThisChar = m_states[ThisState].m_atrans[This_i].symbol;
//         for ( int That_i=0; That_i<nfa.m_states[ThatState].GetSize(); That_i++ )
//         {
//             ThatChar = nfa.m_states[ThatState].m_atrans[That_i].symbol;
//             if ( ThatChar == ThisChar || ThatChar == SYMBOL_STOP || ThatChar == SYMBOL_STOP )
//             {
//                 if ( !inc) mfa_m_states[thatState].m_atrans[that_i].next == -1 ) // found This Fi

```

LASNEA.CPP 3-24-98 11:26a

.. NAL

```

len = strlen(buf);
p += len + 1;
// the first terminated string in buf is firstchars
// skip the first char

size += len + 1;
len = strlen(p);
p += len + 1;
size += len + 1;

int state_count;
state_count = (int)buf[0];
LCTSTR state_offsets = p;

int TotalDistance;
return buf[0] * LCTSTR state_offsets, STATE_BUF(0), 0, 0, TotalDistance;
}

// The "walk" function for the "match" function above. The assertion that buf is divisible
// by 3 shows that the buffer is a series of transition records which have a length of 3.
// Position 0 in each record is the current state (i.e. That_i) Position 1 is the next state
// (i.e. That_i+1). The second symbol is the third byte (That_i+2).
bool ONA::BufMatch(LCTSTR state_offsets, LCTSTR buf_state, TOTAL thisState, TOTAL thisBufState, int TOTALD
    == 0;
{
    int trans_count = (int) buf_state[0];
    buf_state++;
    TOTAL thisChar, bufChar;
    for (int This_i=0; This_i < thisState.GetSize(); This_i++)
    {
        thisChar = (TOTAL) m_states[thisState] m_trans[This_i].symbol;
        for (int Buf_i=0; Buf_i < (trans_count+1); Buf_i++)
        {
            bufChar = (TOTAL) buf_state[Buf_i+1];
            if ( soundDistance Pass(bufChar, thisChar) / bufChar == thisChar / || bufChar == SYMBOL
                == SYMBOL || thisChar == SYMBOL_SYMBOL )
            else
            {
                if (bufChar == thisChar || bufChar == SYMBOL_SYMBOL || thisChar == SYMBOL_SYMBOL )
                {
                    if ( (int) m_states[thisState] m_trans[This_i].next == -1 || found This Fi
                    {
                        if ( (int) buf_state[Buf_i+1] == -1 || found Buf FINAL
                        return TRUE;
                    }
                    else
                    {
                        // This path is done and haven't reached the end of buf path
                        // so go no further down buf path
                    }
                }
                else // more This path to go
                {
                    if ( (int) buf_state[Buf_i+1] == -1 || found Buf FINAL
                    // so go no further
                    else
                    {
                        int next_state = (int) buf_state[Buf_i+1]; // state_offsets
                        LCTSTR next_buf = STATE_BUF(next_state); // state_offsets
                    }
                }
            }
        }
    }
    return (int)state_offsets[i+1];
}

```

Page 5 of 7

```

while(ReadState(LAB, lab, sy))
{
    if (sy == SYMBOL_LABEA)
    {
        m_Lambda = TRUE;
        continue;
    }
    if (LabInverse)
        Add(s1, s0, sy);
    else
        Add(s0, s1, sy);
}
m_Box = pBox;
return TRUE;
}

// trace() is a debugging method which dumps the contents of the
// state table to the debug window and to an output stream.
void ONA::Trace(ostream& out)
{
    string display;
    unsigned s0, s1, sy;
    display.Format("Box: %s\n", m_Box);
    TRACE(display);
    out << display;
    display.Format("First: %s\n", m_FirstChars);
    TRACE(display);
    out << display;
    for (int state=0; state < m_states.GetSize(); state++)
    {
        for (int trans=0; trans < m_trans[state].GetSize(); trans++)
        {
            s0 = state;
            s1 = m_states[state] m_trans[trans].next;
            sy = m_states[state] m_trans[trans].symbol;
            display.Format("%d : %d : %s\n", s0, s1, sy);
            TRACE(display);
            out << display;
        }
    }
}

if (int len;
LCTSTR p = buf;

```

LASNFA.CPP 3-24-98 11:24a

```

// Subsetting match - any end point returns true
return TRUE;
}
else // more than this path to go
{
    if ( (int) buf_state(buf_i-1) == -1 ) // found buf FIMOL
    {
        // Subsetting match - any end point returns true
        return FALSE;
        // no go further
    }
    else
    {
        int next_state = (int) buf_state(buf_i-1);
        LPCTSTR next_buf = STATE_BUF(next_state); // state_offset
        if (next_state == 0)
        {
            if ( SubBufMatch( state_offsets, next_buf,
                             m_states(ThisState), m_aTrans(This_i).next,
                             next_state ) )
                return TRUE;
        }
    }
}
} // endif ThisChar==BufChar
else
    continue;
}
} // endif ThisChar==BufChar
return FALSE; // didn't get out of this state
}
// This function inserts on state table into another. This creates a backward
// flowing state table so it currently can't be used in a naive search. The naive
// search routine should be modified to allow these kinds of tables
//
// BOOL CHIA::InsertHalt(int start, int stop, Offset nfa)
// {
//     TCHAR sz_s1[50];
//     m_MaxState = m_states.GetSize();
//     for( int state=0; state < nfa.m_states.GetSize(); state++)
//     {
//         sy = nfa.m_states[state].m_aTrans[trans].symbol;
//         if ( state == 0 )
//         {
//             AddFirstChars(sy);
//             sz = start;
//         }
//         else
//         {
//             sz = state + m_MaxState;
//             if ( nfa.m_states[state].m_aTrans[trans].next == FIMOL )
//             {
//                 AddFirstChars(sy);
//                 sz = stop;
//             }
//             else
//             {
//                 sz = nfa.m_states[state].m_aTrans[trans].next + m_MaxState;
//             }
//         }
//     }
// }

```

Page 6 of 7

```

if (next_state == 0)
{
    if ( SubBufMatch( state_offsets, next_buf,
                     m_states(ThisState), m_aTrans(This_i).next,
                     next_state, TotalDistance ) )
        return TRUE;
}
} // endif ThisChar==BufChar
} // endif ThisChar==BufChar
return FALSE; // didn't get out of this state
}
// This function inserts on state table into another. This creates a backward
// flowing state table so it currently can't be used in a naive search. The naive
// search routine should be modified to allow these kinds of tables
//
// BOOL CHIA::InsertHalt(int start, int stop, Offset nfa)
// {
//     TCHAR sz_s1[50];
//     m_MaxState = m_states.GetSize();
//     for( int state=0; state < nfa.m_states.GetSize(); state++)
//     {
//         sy = nfa.m_states[state].m_aTrans[trans].symbol;
//         if ( state == 0 )
//         {
//             AddFirstChars(sy);
//             sz = start;
//         }
//         else
//         {
//             sz = state + m_MaxState;
//             if ( nfa.m_states[state].m_aTrans[trans].next == FIMOL )
//             {
//                 AddFirstChars(sy);
//                 sz = stop;
//             }
//             else
//             {
//                 sz = nfa.m_states[state].m_aTrans[trans].next + m_MaxState;
//             }
//         }
//     }
// }

```

LASNFA.CPP 3-24-98 11:24a

```

stream.put((char)0);
stream.write( nfa.m_sourceName, nfa.m_sourceName.GetLength() );
stream.put((char)0);

// write number of states
state_count = nfa.m_states.GetSize();
stream.put( (char) state_count );

// write offset to each state
for( offset = state_count+1; s0;
    {
        stream.put((char) offset);
        int trans_count = nfa.m_states[s].m_aTrans.GetSize();
        offset += ((3*trans_count)+1);
    }

// now continue with the actual transitions
for( s0; s0 < nfa.m_states.GetSize(); s++ )
    {
        // write transition count before each list of transitions
        //
        stream.put( (char) nfa.m_states[s].m_aTrans.GetSize() );
        for ( t=0; t < nfa.m_states[s].m_aTrans.GetSize(); t++ )
            {
                stream.put((char) s );
                stream.put( (char) nfa.m_states[s].m_aTrans[t].next );
                stream.put( (unsigned char) nfa.m_states[s].m_aTrans[t].symbol );
            }
        stream.put((char)254);
    }
return stream;
}

```

```

Add( s0, st, sy );
Cycling display;
display.Format("%ld : %ld : %c\n", s0, st, sy);
}
return TRUE;
}

// The method LightStateTableTraverseal walks all the paths
// within the state table. It just counts the number of paths,
// and does not capture the variants that could be generated
// The parameter chr is really just for recursion so it is defaulted
// to NULL for the first call
int OFA::LightStateTableTraverseal(TCHAR state, char chr /* = NULL */)
{
    static int numVariants = 0;

    // don't walk if OFA not valid or something interrupted
    // the OFA processing - check is m_source still necessary?
    if (m_source == FALSE || m_isOpen == FALSE)
    {
        // this is the first time through, so clear out the
        // transition table. Also, check to see if
        // there is a blank file etc. which indicates
        // that OFA is not included in the state tables, so
        // we must pick it off separately.
        if (chr == NULL)
        {
            numVariants = 0;
            if (m_blank)
            {
                m_variance.Add(1);
            }
        }
        char ThisChar;
        for ( int i=0; i<m_states.GetSize(); i++ )
        {
            ThisChar = (char) m_states[state].m_aTrans[i].symbol;
            if ( (int) (m_states[state].m_aTrans[i].next) == -1 )
            {
                numVariants++;
            }
            else if( state != m_states[state].m_aTrans[i].next )
            {
                LightStateTableTraverseal( m_states[state].m_aTrans[i].next, ThisChar );
            }
        }
        return numVariants;
    }
    int ONA::GetNumVariants()
    {
        return LightStateTableTraverseal(0);
    }
}

// The operator which actually writes a state table to a file or
// string stream. Character 255 (that's already in symbol) is used
// to ID the FINAL state and character 254 is used as the record delimiter.
// note: the first two items in the record are zero delimited strings: the rest
// of the record is the state table in 3 byte records.
ostream& operator<< (ostream& stream, OFA& nfa)
{
    int s.t., offset;
    int state_count =0;
    stream.write( nfa.m_FirstChar, nfa.m_FirstChar.GetLength() );
}

```



```
//
//
#include "Header.h"

// Send the appropriate data to reader
//
// narrative paragraph number 4.2.4
// narrative paragraph number 4.3.3
void Header::submitToReader(Header &reader)
{
    ExclName *tempExclName1 = new ExclName(nameString, editDistScore(0),
        startConsonants(0),
        startVowelFlag(0),
        exactMatchFlag == 'Y',
        nameCode,
        culture(0), // class culture
        culture(0)); // pipe culture

    if (nameCultures == 1)
        reader.submit(tempExclName1);
    else
        ExclName *tempExclName2 = new ExclName(nameString, editDistScore(1),
            startConsonants(1),
            startVowelFlag(1),
            exactMatchFlag == 'Y',
            nameCode,
            culture(1), // class culture
            culture(1)); // pipe culture

    reader.submit(tempExclName1, tempExclName2);
}
```



```

    matchingRuleIndex = matchingRuleIndexes(i);
    if (matchingRuleIndex != -1)
    {
        simplifyBeginnings(matchingRuleIndex);
    }
}

// The main working method for the RuleSet class. FindRule() is passed
// an new (word) and an index into it and it finds a rule which passes
// all the contexts for that index (letter!). The output is then added
// to the return string parameter. If no rule is found for a letter then
// the lowercase form of the letter is added to the return string. The actual
// rule (word) is not modified by this routine, therefore, letters replaced
// by a rule are still available to the context portions of other rules. Since
// the scanning of a word proceeds left to right only one match context of a rule
// applies to any letter.
//
//
int RuleSet::FindRule(LPCSTR word, int index, CString *psetString)
{
    extern ostream err;
    ASSERT(m_Rules.size() > 0);
    BOOL MatchedRule = FALSE;
    CString matched_part;
    int count, remainder;
    for (int rule_index=0; rule_index < m_Rules.size(); rule_index++)
    {
        MatchedRule = FALSE;
        count=0;
        if (m_Rules[rule_index].match m_RuleIndex)
        {
            MatchedRule = TRUE;
            remainder = index;
        }
        else if (m_Rules[rule_index].match Match(word,index), count )
        {
            matched_part = word(index);
            matched_part = matched_part.Left(count);
            remainder = index + count;
            if ( remainder >= strlen(word) )
            {
                remainder = strlen(word)-1;
            }
            else
            {
                continue;
            }
        }
        if (m_Rules[rule_index].left.Match(word, index, count) )
        {
            // do rest of the loop body
        }
        else
        {
            continue;
        }
        if (m_Rules[rule_index].right.Match( word[remainder], count ) )
        {
            // do rest of the loop body
        }
        else
        {
            continue;
        }
        // Check for context variables. The output of a rule can contain
        // variables like $1 meaning the first letter matched in the match
        // context this variable is simply replace by the letter before it
        // is attached to the output string. Otherwise the output is con-
        // catenated as-is.
        //

```

PARSE.CPP 3-24-98 11:24a

```

    if ( strstr(m_Rules[rule_index].output, '$') == NULL )
    {
        psetString += m_Rules[rule_index].output;
    }
    else
    {
        AddressIndex("in else");
        int iTag, pos;
        CString tag;
        for ( iTag=1; iTag <= matched_part.GetLength(); iTag++)
        {
            tag.Format("%d", iTag);
            pos = m_Rules[rule_index].output.Find(tag);
            if ( pos == -1 )
            {
                break;
            }
            psetString += m_Rules[rule_index].output.Left(pos);
            psetString += matched_part(iTag-1);
            psetString += m_Rules[rule_index].output.Mid(pos+2);
        }
    }
    return remainder;
}

return index+1;
}

// The main working method for the RuleSet class. FindRule() is passed
// a new (word) and an index (index) into that word. It finds a rule
// which passes all the contexts (left, right and match) for the
// substring starting at that index. The output associated with the
// selected rule is then appended to the return string (psetString).
// If no rule is found that matches the substring, the first character
// of the substring is converted to lowercase and added to the return string,
// and the index is incremented by 1.
// The actual new (word) is not modified by this routine. This allows
// subsequent calls to this function to be able to use characters that were
// already matched (i.e. in the evaluation left and right contexts).
// In general, once a rule is found, the index is incremented by the number
// of characters that participated in the matching context, and the function
// returns, noting the index of the rule that was used (matchingRuleIndex).
// However, some rules contain matching contexts that can evaluate to NULL.
// will match anything. In these cases, it is possible that the selected rule
// will cause output to be appended, but will not increment the index variable.
// When this happens, the function continues looking for a rule that will
// "eat up" at least one character from the word being worked on. Thus, it
// is possible that the function will fire more than one rule. For this reason,
// we pass in a vector of rule indexes, rather than a single rule index.
int RuleSet::FindRule(LPCSTR word, int wordIndex, CString *psetString,
    vector<int> *matchingR

```

Page 2 of 17

```

else
{
    int iTag, pos;
    String tag;
    String matchedPortion;

    // portion
    matchedPortion = word[wordIndex];
    matchedPortion = matchedPortion.Left(matchCharCount);
    for ( iTag=1; iTag <= matchedPortion.GetLength(0); iTag++ )
    {
        tag.Format("%d", iTag);
        pos = m_Rules[ruleIndex].output.Find(tag);
        if ( pos == -1 )
            break;
        *pStrString += m_Rules[ruleIndex].output.Left(pos);
        *pStrString += matchedPortion(iTag-1);
        *pStrString += m_Rules[ruleIndex].output.Mid(pos-2);
    }

    // if we are here, we applied a rule, so note which one was used
    matchingRuleIndex = vector_push_back(rulesIndex);

    // if we actually ate up a character, we are done, so break the loop
    // If we matched without eating a character, keep looking for a rule
    if ( matchCharCount != 0 )
    {
        foundRuleThatAteCharacter = true;
        break;
    }
}

// if we could not find a rule that ate a character, just
// copy the character (lowercase) to the return string and
// advance the remainder to the next character in the word.
if ( foundRuleThatAteCharacter == false )
{
    *pStrString += tolower(word[wordIndex]);
    remainder = wordIndex + 1;
}

return remainder;
}

// Passed a rule line from the input file, this function parses
// the line and checks for some basic errors. Finally it creates
// a rule object and adds it to the array. The output stream err
// is the error log and is opened in CppApp.cpp. This is a simple
// implementation of the rule grammar if the rules become more complicated
// an actual grammar generator (like yacc) should be used.

BOOL ruleSet::AddRule(CString strRule)
{
    extern ofstream err;

    BOOL bRet = TRUE;
    BOOL b1, b2, b3, b4;
    b1=b2=b3=b4=FALSE;
    CString strSaveLine(strRule);
    CString delims("\\t");

    int nCount = 0, nPos, nItem = 0;

```

```

// when matching the left and right contexts. We do not
// care about the number of matching chars in these cases.
foundRuleThatAteCharacter;
wordLen = strlen(word);

// clear out vector that holds the index of the rules that are fired during this
// call function.
matchingRuleIndexVector.clear();

// note that we have not found a rule to eat up the character(s) at the
// beginning of the portion of the word we are looking at.
foundRuleThatAteCharacter = false;

// loop through rules, looking for one to use
for (int ruleIndex = 0; ruleIndex < m_Rules.size(); ruleIndex++)
{
    matchedCharCount = 0;
    if (m_Rules[ruleIndex].match.Match(word[wordIndex], matchCharCount))
    {
        // we got a match on the matching context, so look at the
        // left context. Note that we pass in tempMatchCount since we do not
        // care about how many characters are involved in the left context.
        if (m_Rules[ruleIndex].left.LeftMatch(word, wordIndex, tempMatchCount))
        {
            // we got a match on the left context, so look at the
            // right context. Note that we pass in tempMatchCount since we do not
            // care about how many characters are involved in the right context.
            // Also note that we match the match at word[remainder], so that
            // we are looking at the character to the right of the characters
            // that were involved in the matching context match.
            // note how many characters were eaten up by the match context match
            // and increment the remainder by that much. We will return the remainder
            // at the end of the function. The remainder is also needed so that we
            // know where to begin the right context match.
            remainder = wordIndex + matchCharCount;

            // I think this code will be a problem if the word is not padded at the end
            // a space (it currently is). Consider a single character word that matches
            // character. When the function comes to it, the index is 0, and the remainder
            // becomes 1, which is equal to strlen(word). The remainder then gets so
            // to 0 by the conditional code, which would put us into an infinite loop
            // always calling this function looking at the first character.
            if (remainder >= wordLen)
                remainder = wordLen - 1;

            if (m_Rules[ruleIndex].right.Match(word[remainder], tempMatchCount))
            {
                // we have found a rule that matches all three contexts, so
                // Append the rule's output portion to the pStrString variable.
                // Check for context variables. The output of a rule can contain
                // variables like $1 meaning the first letter matched in the match
                // context this variable is simply replace by the letter before it
                // is attached to the output string. Otherwise the output is on
                // estimated as-is.
                if (m_Rules[ruleIndex].getDoesOutputContainContext() == false)
                    *pStrString += m_Rules[ruleIndex].output;
            }

```

```

case '(': paren++; break;
case '[': bracket++; break;
case '{': has_op = TRUE;
}
p++;
}
if (paren != 0)
{
err << "LINE: " << m_line << " " << (LPCSTR)strSaveline << endl;
err << " Unbalanced parenthesis " << endl;
bset = FALSE;
}
if (bracket != 0)
{
err << "LINE: " << m_line << " " << (LPCSTR)strSaveline << endl;
err << " Unbalanced brackets " << endl;
bset = FALSE;
}
if (check == 3)
{
if (has_op && !has_paren)
{
err << "LINE: " << m_line << " " << (LPCSTR)strSaveline << endl;
err << " Output has Binary operator | without parenthesis " << endl;
bset = FALSE;
}
}
// no need to go on if there's an error
if (!bset)
return bset;
} // further syntax checking on 4 columns
// this is a KLUDGE. The rules define silent as () sometimes.
// However, () is not a valid regular expression, either by itself
// or when used as part of a larger regular expression. We therefore
// have special code to translate it to an empty string.
// Note that it is defined as () in the rules because the simplified
// rules needs to have some sort of token for each output string,
// and a NULL wont work (NULL is not a token). An alternative to this
// KLUDGE is to:
// Replace the "()" in the rules with NULL (**). Then either
// changed the code that reads in the simplified rules to
// work off of position instead of tokens, or assign a special
// token for silent (just in the simplified rules).
if (output == "()")
output = "";
rule_rule(left_match,right_output);
if (rule_m_bad)
{
err << "LINE: " << m_line << " " << (LPCSTR)strSaveline << endl;
err << " Problem with regular expression " << endl;
bset = FALSE;
}
else
m_Rules.push_back(rule);
return bset;
}
RuleSet::RuleSet()
{
m_bCompiled = FALSE;
}

```

```

Exiting stream:
Exiting left_match.right_output:
while (m_Count < 4) // 4 items
strList.TrimLeft();
if ((m_pos = strList.FindOneOf(delims)) >= 0)
strItem = strList.Left(m_pos);
else
strItem = strList;
switch (m_Count++)
{
case 0: if (!CheckSymbol(strItem,left)) bset=FALSE; break;
case 1: if (!CheckSymbol(strItem,right)) bset=FALSE; break;
case 2: if (!CheckSymbol(strItem,right)) bset=FALSE; break;
case 3: if (!CheckSymbol(strItem,output)) bset=FALSE; break;
}
strList = strList.Mid(m_pos+1);
}
if (!b1)
{
err << "LINE: " << m_line << " " << (LPCSTR)strSaveline << endl;
err << " Problem with left portion of rule " << endl;
}
if (!b2)
{
err << "LINE: " << m_line << " " << (LPCSTR)strSaveline << endl;
err << " Problem with match portion of rule " << endl;
}
if (!b3)
{
err << "LINE: " << m_line << " " << (LPCSTR)strSaveline << endl;
err << " Problem with right portion of rule " << endl;
}
if (!b4)
{
err << "LINE: " << m_line << " " << (LPCSTR)strSaveline << endl;
err << " Problem with output portion of rule " << endl;
}
// no need to go on if theres an error
if (!bset)
return bset;
for (int check=0; check < 4; check++)
{
LPCSTR p;
p = NULL;
// Check for further syntax errors
switch(check)
{
case 0: p = left; break;
case 1: p = match; break;
case 2: p = right; break;
case 3: p = output; break;
}
int paren=0,bracket=0;
bool has_op=FALSE, has_paren=FALSE;
while (*p)
{
switch(*p)
{
case '(': paren++; has_paren = TRUE; break;
case '[': bracket++; break;
}
}

```

```

Rule::Rule()
{
    m_Bad = FALSE;
}

// reset the RuleSet so that it can be re-used
void RuleSet::ResetAll()
{
    m_Rules.clear();
    m_Symbols.clear();
    m_Compiled = FALSE;
}

// add a symbol manually from the two parameters. This function can
// be used to hard code symbols into the rule set. Currently all symbols
// are added from the rule sets.
void RuleSet::AddSymbol(LPCSTR name, LPCSTR value)
{
    CString temp = name;
    Cap,LowerFirst();
    m_Symbols[temp] = value;
}

// This method parses a Set command from the input file and adds the
// symbol to the symbol set. A set command is simply three columns delimited
// by white space. Set, variable name, "string". This is a simple
// implementation of the rule grammar. If the rules become more complicated
// an actual grammar generator (like yacc) should be used.
BOOL RuleSet::AddSymbol(LPCSTR symbolName)
{
    BOOL bSet = FALSE;
    char *delim = " \t";
    char symbol[MAX_SYMBOL_LEN + 1];
    char value[MAX_VALUE_LEN + 1];
    const char *valuePtr;
    const char *symbolPtr;
    int i;
    int valueLen;
    int symbolLen;
    int valuePos;
    int symbolPos;

    // since we already stripped this before we came into the function,
    // the first white space is the white space between the SET and
    // the symbol. We have already checked for the set, so we can
    // begin at the fourth character.
    symbolPtr = symbolName + 3;
    valuePos = symbolName + 3;
    whiteSpaceLen = strlen(symbolPtr, delim); // get rid of leading whitespace
    symbolLen = strlen(symbolPtr, delim);
    whiteSpaceLen = strlen(symbolPtr, delim);
    if (symbolLen > 0)
    {
        // now copy the symbol into a separate buffer
        strcpy(symbol, symbolPtr, symbolLen);
        symbol[symbolLen] = EOS;

        valuePtr = symbolPtr + symbolLen;
        whiteSpaceLen = strlen(valuePtr, delim); // get rid of leading whitespace
        // now copy the value into a separate buffer
        // The value could be a quoted value (possibly with embedded spaces),
        // or a single string value (without spaces). We must therefore see if
        // we are dealing with a quoted value. Otherwise, if we just look for

```

PARSE.CPP 3-24-98 11:24a

```

// white space, we will get confused with the embedded whitespace that
// is part of the value.
valuePtr = whiteSpaceLen;
if ((*valuePtr) == '\\')
{
    char *endQuotePtr = strchr(valuePtr + 1, '\\');
    if (endQuotePtr != NULL)
    {
        // grab everything between the quotes
        valueLen = (endQuotePtr - valuePtr) - 1;
        strcpy(value, valuePtr + 1, valueLen);
        value[valueLen] = EOS;
    }
    else
    {
        err << "LINE: " << m_Line << " << (LPCSTR)symbolsLine << endl;
        err << "Missing second quote after SET" << endl;
    }
}
else
{
    valueLen = strlen(valuePtr, delim);
    strcpy(value, valuePtr, valueLen);
    value[valueLen] = EOS;
}

// make sure we have a symbol. The value can be empty
if (symbol[0] != EOS)
{
    AddSymbol(symbol, value);
    bSet = TRUE;
}
}

if (bSet == FALSE)
{
    err << "LINE: " << m_Line << " << (LPCSTR)symbolsLine << endl;
    err << "Invalid SET syntax" << endl;
}

return bSet;
}

// This function searches a CString object to see if it contains any embedded
// symbols. A symbol is embedded in a string using the {}= delimiters.
// This is a simple implementation of the rule grammar if the rules become
// more complicated an actual grammar generator (like yacc) should be used.
//
// BOOL RuleSet::CheckSubSymbol(CString target, CString& RetString)
{
    extern ostream err;
    BOOL bSet = TRUE;
    CString tmpString;
    RetString = target;
    int ipos, rpos;
    while ( ( ipos = target.Find('(') ) != -1 )
    {
        if ( ( rpos = target.Find(')') ) == -1 )
        {
            err << "LINE: " << m_Line << endl;
            err << "Missing ')-' in " << (LPCSTR)target << endl;
            bSet = FALSE;
            break;
        }
        if ( ( rpos < ipos ) )
        {
            err << "LINE: " << m_Line << "Mis-placed ')-' in " << (LPCSTR)target << endl;

```

Page 5 of 17

Page 8 of 17

PARSE.CPP 3-24-98 11:24a


```

// clear out our member variable that organizes the replacement
// strings by code. Also, push a dummy blank or null entry onto
// the vector so that the indexing will match the 1 based scheme
// we use for codes.
m_replacementBeginStringVec.clear();
m_replacementBeginStringVec.push_back("");
m_nardistinctStrings = 0;

for (replacementSetIter = replacements.begin(); replacementSetIter != replacements.end(); re
    placementSetIter++)
{
    m_nardistinctStrings++;
    // now see how many characters this regex can produce
    replacementSetIter->RemoveAll();
    replacementSetIter->Set(replacementSetIter);
    replacementSetIter->Push(0, JustStr);
    namVariantsForThisRegex = replacementSetIter->m_variants.GetSize();
    longestVariantForThisRegex = 0;
    for (varIndex = 0; varIndex < namVariantsForThisRegex; varIndex++)
    {
        length = replacementSetIter->m_variants(varIndex).GetLength();
        if (length > longestVariantForThisRegex)
            longestVariantForThisRegex = length;
    }
    // store the longestVariantForThisRegex and
    // namVariantsForThisRegex values in our vectors.
    // this will help reduce the number of names we will
    // have to create NPAs for.
    longestVarVec.push_back(longestVariantForThisRegex);
    namVarVec.push_back(namVariantsForThisRegex);
    replacementBeginStringVec.push_back((LPCSTR)"replacementSetIter");

    // now see if this regex is actually the same as a regex we
    // have already processed. If it is, it must have the same
    // number of variants and its longest variant must be the
    // same length as the other. We do these quick checks to
    // avoid the costly O(n) processing
    // Note that we only need to look from the start of the
    // set to our current position.
    int vecIndex = 0;
    BOOL isSame = FALSE;
    for (replacementSetIter2 = replacements.begin(); replacementSetIter2 != replacementSe
        tIter; replacementSetIter2++)
    {
        if ((longestVarVec[vecIndex] == longestVariantForThisRegex) && namVarVec[vecIndex] == namVariantsForThisRegex)
        {
            // now we need to see if every variant from the new regex
            // is found in the variants generated by the regex of the
            // possibly equivalent regex that happens to have the
            // same number of variants and the same max length.
            // Note that because they have the same number of variants,
            // we only need to check in one direction. That is, we
            // only need to make sure what's in regex1's variant
            // array is also in regex2's variant list, and not
            // vice-versa.
            posequivita.RemoveAll();
            posequivita.Set(replacementBeginStringVec[vecIndex]);
            posequivita.RemoveAll(0, JustStr);
            // Line below should be unnecessary, because we should already
            // know that the possible equivalent regular express has the
            // same number of variants as our candidate regex. namVariantFor
            // thisRegex = posequivita.m_variants.GetSize();
        }
    }
}

```

```

BOOL foundMatchForThisVariant;
for (int outerMatchIndex = 0; outerMatchIndex < numVariantsPossEqvita
    foundMatchForThisVariant = FALSE;
    for (int innerMatchIndex = 0; innerMatchIndex < namVariantPos
        if (replacementSetIter->m_variants(outerMatchIndex) == pos
            foundMatchForThisVariant = TRUE;
            break;
        }
    }
    // went all the way through the inner (posequivita) NPA,
    // find the variant from the outer (replacementSetIter) NPA,
    // if (foundMatchForThisVariant == FALSE) {
        break;
    }
    // after the above nested for loops, the variable foundMatchFor
    // should be FALSE if the NPAs are not equivalent. If the variab
    // TRUE, it means every variant in replacementSetIter was found in
    // posequivita, so they are the same.
    if (foundMatchForThisVariant == TRUE) {
        // since they are the same, we assign a code that is the
        // same as the ordinal position of the item we were compa
        // it to (vecIndex).
        codeForThisRegex = regCodeVec[vecIndex];
        isSame = TRUE;
        break;
    }
    vecIndex++;
    // was not the same, so assign it the next available code.
    // and bump up the next avail code variable.
    if (isSame == FALSE) {
        // save a pointer to the replacement string for this code. This
        // vector is indexed by the code, so that the string at the
        // say, third item is the string for code 3. This works, because we
        // push a blank onto the vector at the beginning since vectors are
        // 0 indexed, and the codes start at 1.
        m_replacementBeginStringVec.push_back((LPCSTR)"replacementSetIter");
        codeForThisRegex = nextAvailCode;
        nextAvailCode++;
    }
    regCodeVec.push_back(codeForThisRegex);
    m_nardistinctCodes = nextAvailCode - 1; // since we did not assign it.

    // now we have a vector of regexes - replacementBeginStringVec
    // and a parallel vector of regex codes - regCodeVec.
    // We use these to go back to original rules in the rules
    // set and assign the right code to the ruleSet's member
    // variable m_replacementBeginStringVec. This is necessary because the
    // vector m_replacementBeginStringVec is not in the same order as the rules
    // in the rule set. We use the replacementBeginStringVec to
    // find the rule we are looking for, and then assign

```

```

        if (i == j)
            recCodeComparisonArray[i][j] = 100;
        else
            recCodeComparisonArray[i][j] = CalculateCodeCompScore(i, variantsForCode[i]);
    }
    delete variantsForCode;
}

return rc;
}

// this is the new float version of this function
// the algorithm to generate the RC comparison values will now
// be based on the edit distance score between the two RDNs instead
// of the digraph analysis of the two RDNs
// this function has exception handling
BOOL RuleSet::CreateCodeComparisonArray()
{
    BOOL rc = TRUE;
    const char *regForCode;
    int i, j;
    tcd_vargen_code genVarRC;

    //File matrixfile;
    //char *matrix_file_name = "reanatrix.bin";
    ifstream matrixfile("reanatrix.bin");
    if (!matrixfile.is_open())
    {
        //std::cerr << "The reanatrix.bin file was not found!\n";
        // declare the Cpptrix object here
        Cpptrix RecApprox("floatdist.rul");
        // this function has exception handling
        // initialize the recCodeComparisonArray to all 1.0
        for (i = 0; i < 256; i++)
            for (j = 0; j < 256; j++)
                recCodeComparisonArray[i][j] = 1.0; // 1.0 = default no match

        // remember to go from 1 to numCodes inclusive, since
        // codes begin at 1, not 0.
        for (i = 1; i <= numDistinctCodes; i++)
        {
            regForCode = GetRegForCode(i);
            if (regForCode == NULL)
            {
                char errMsg[1000];
                sprintf(errMsg, "Could not find regex for code %d", i);
                MessageBox(errMsg);
                break;
            }
            vector<CString> *variantsForCode = genVariants(regForCode, FALSE, FALSE, 0, genVarRC);
            for (j = 1; j <= numDistinctCodes; j++)
            {
                if (i == j) // recCodeComparisonArray is of floats
                    recCodeComparisonArray[i][j] = 0.0; // 0.0 means that i and j have no differences
                else
                    recCodeComparisonArray[i][j] = CalculateCodeCompScore(i, variantsForCode[j], RecApprox);
            }
            delete variantsForCode;
        }
        WriteReanatrixToFile(); // this writes the rec matrix file
    }
}

```

Page 11 of 17

```

// the RSC to the a_regCodeVec at that position.
int
numRules = m_rules.size();
// clear out the previous values for m_regCodeVec
// and set each value to 0;
m_regCodeVec.clear();
for (int j = 0; j < numRules; j++)
{
    m_regCodeVec.push_back(0);
}

for (int i = 0; i < numDistinctCodes; i++)
{
    for (int j = 0; j < numRules; j++)
    {
        if (replacementRegForCodeVec[i] == m_rules[j].output)
        {
            // found the rule, so assign the code
            // but don't break, because the replacement
            // string might exist more than once - remember,
            // they are not deduped.
            m_regCodeVec[j] = regCodeVec[i];
            // m_regCodeVec[i] = regCodeVec[i];
        }
    }
}

// now use the codes we just assigned to build the
// table that says how closely two codes are related
if (buildRSCComArray)
{
    CreateCodeComparisonArray();
}

delete replacements;
else
{
    rc = FALSE;
}

return rc;
}

// function to encode the rules by determining the unique set of
// replacement regions, and assigning a unique code to each one.
// regions that are equivalent are given the same code.
// The function makes sure that the rules have already been compiled.
BOOL RuleSet::CreateCodeComparisonArray()
{
    BOOL rc;
    const char *regForCode;

    // remember to go from 1 to numCodes inclusive, since
    // codes begin at 1, not 0.
    for (int i = 1; i <= numDistinctCodes; i++)
    {
        regForCode = GetRegForCode(i);
        if (regForCode == NULL)
        {
            char errMsg[1000];
            sprintf(errMsg, "Could not find regex for code %d", i);
            MessageBox(errMsg);
            break;
        }
        vector<CString> *variantsForCode = genVariants(regForCode, FALSE, FALSE, 0);
        for (int j = 1; j <= numDistinctCodes; j++)
    }
}

```

PARSE.CPP 3-24-98 11:24a

```

else
{
    ReadBackMatrixFromFile(matrixFile);
}
return rc;
}

void RuleSet::WriteBackMatrixToFile()
{
    int i, j;
    ofstream o_file("recmatrix.bin");
    for(i = 0; i < m_numDistinctCodes; i++)
    {
        for(j = 0; j < m_numDistinctCodes; j++)
        {
            o_file << recCodeComparisonArray[i][j] << endl;
        }
        o_file.close();
    }

    void RuleSet::ReadBackMatrixFromFile(ifstream &matrixFile)
    {
        int i, j;
        float element;
        unsigned char buf[BUFFSIZE];

        for(i = 0; i < m_numDistinctCodes; i++)
        {
            for(j = 0; j < m_numDistinctCodes; j++)
            {
                matrixFile->getline(char*buf, BUFFSIZE);
                element = atof(const char*buf);
                recCodeComparisonArray[i][j] = element;
            }
        }

        // RuleSet::CreateCodeComparisonArray()
        {
            bool rc;
            const char *regBufForCode1;
            int i, j;

            //ifstream matrixFile;
            // for some reason the ios::nocreate flag is not working
            //matrixFile.open("recmatrix.bin", ios::binary, filebuf::at_none);

            TRY
            {
                CFFile mtrxFile("recmatrix.bin", CFFile::modeRead);
            }
            CATCH (CFFileException, e)
            {
                if(e-m_cause == CFFileException::fileNotFound)
                    afxMessageBox("The recmatrix.bin file was not found!");
            }

            END_CATCH

            // declare the Cpprox object here

```

PARSE.CPP 3-24-98 11:24a

```

OApprox RegApprox("floadist.nul");

// Initialize the recCodeComparisonArray to all 1.0
for(i = 0; i < 256; i++)
for(j = 0; j < 256; j++)
    recCodeComparisonArray[i][j] = 1.0; // 1.0 = default no match

// remember to go from 1 to numCodes inclusive, since
// codes begin at 1, not 0.
for(i = 1; i <= m_numDistinctCodes; i++)
{
    regBufForCode1 = GetRegBufForCode(i);
    if (regBufForCode1 == NULL)
    {
        char arrRegBuf[1000];
        sprintf(arrReg, "Could not find regbuf for code %d", i);
        AfxMessageBox(arrReg);
        break;
    }
    vector<CString> variantsForCode1 = genVariants(regBufForCode1, FALSE, FALSE, 0);
    for(j = 1; j <= m_numDistinctCodes; j++)
    {
        if (i == j) // recCodeComparisonArray is of floats
            recCodeComparisonArray[i][j] = 0.0; // 0.0 means that i and j have no difference
        else
            recCodeComparisonArray[i][j] = CalculateCodeCompScore(i, variantsForCode1, &vec);
        // print();
    }
    delete variantsForCode1;

    //AfxMessageBox("rec comparison matrix built");
    //ifstream f_out("recmatrix.txt");
    //ostringstream f_out;
    //f_out.close();
    //AfxMessageBox("rec matrix dumped");

    return rc;
}

//
//
// function to search for the recCode in our vector of
// recCodes for these rules, and return the corresponding
// regbuf string. Return NULL if the code is not found.
const char * RuleSet::GetRegBufForCode(int recCode)
{
    if ((recCode < 1) || (recCode > m_ReplacementRegBufStringVec.size()))
        return NULL;
    else
        return (LPTSTR)m_ReplacementRegBufStringVec[recCode];

    int numRules = m_ReplacementRegBufVec.size();
    const char *returnCharPtr = NULL;
    for (int i = 0; i < numRules; i++)
    {
        if (m_ReplacementRegBufVec[i] == recCode)
        {
            returnCharPtr = m_Rules[i].output;
            break;
        }
    }
    return returnCharPtr;
}

```

Page 12 of 17

```

/* unsigned char Ruleset::ClickCodeCompare(int code2, vector<CString>
{
    const char Ruleset::ClickCodeCompare(int code2, vector<CString>
    unsigned short int returnScore = 0;
    unsigned short int tempScore;

    // get the regex for the second code, so we can compare them
    regForCode2 = GetRegForCode(code2);

    if (regForCode2 == NULL)
    {
        cout << "Error: Could not find regex for code " << code2 << endl;
        return false;
    }
    else
    {
        // generate the variant for each code's regex
        vector<CString> variantsForCode2 = genVariants(regForCode2, FALSE, FALSE, 0);

        // get the counts for the variant vectors
        int numVariantsForCode1 = variantsForCode1->size();
        int numVariantsForCode2 = variantsForCode2->size();

        // now compare the variants in a cartesian product style to
        // determine how close these expressions are.
        for (int i = 0; i < numVariantsForCode1; i++) {
            for (int j = 0; j < numVariantsForCode2; j++) {
                tempScore = digraph_score((variantsForCode1[i], (variantsForCode2[j]));
                if (tempScore > returnScore)
                    returnScore = tempScore;
            }
        }
        delete variantsForCode2;
    }
    return returnScore;
}

// This is the float version of this function
// it uses the edit distance algorithm to calculate the
// comparison value of two rexs rather than the digraph
// score.
float Ruleset::ClickCodeCompare(int code2,
    vector<CString> variantsForCode1,
    vector<CString> variantsForCode2,
    unsigned char genVarRC)
{
    const char regForCode2;
    float returnScore = 1.0;
    float tempScore = 0.0;

    const char variant1;
    const char variant2;
    tds_vargen_code genVarRC;

    // get the regex for the second code, so we can compare them
    regForCode2 = GetRegForCode(code2);

    if (regForCode2 == NULL)
    {
        cout << "Error: Could not find regex for code " << code2 << endl;
        return false;
    }
}

```

```

}
else
{
    // generate the variant for each code's regex
    vector<CString> variantsForCode1 = genVariants(regForCode1, FALSE, FALSE, 0, genVarRC);

    // get the counts for the variant vectors
    int numVariantsForCode1 = variantsForCode1->size();
    int numVariantsForCode2 = variantsForCode2->size();

    // now compare the variants in a cartesian product style to
    // determine how close these expressions are.
    for (int i = 0; i < numVariantsForCode1; i++)
    {
        variant1 = (LPCTSTR)(variantsForCode1[i]);
        for (int j = 0; j < numVariantsForCode2; j++)
        {
            variant2 = (LPCTSTR)(variantsForCode2[j]);
            // here we should be calling the edit distance algorithm
            tempScore = edit_distance_score((unsigned char *)variant1,
                (unsigned char *)variant2,
                &tempScorePtr);
            if (tempScore < returnScore)
                returnScore = tempScore;
        }
        delete variantsForCode2;
    }
    return returnScore;
}

float edit_distance_score(const unsigned char *string1,
    const unsigned char *string2,
    unsigned short *tempScorePtr)
{
    float score = 0.0;

    &tempScorePtr = &gen_float_differences(string1, string2, score);

    return score;
}

/* digraph_score
    A value from 0 to 100 is calculated based on the number of
    digraphs which match between the two given strings.

    Notes:
    The routine ensures that a digraph can only participate in a
    match once.

    Each match results in two points being added to the total. The
    final score is the total number of points divided by the number
    of digraphs that could have matched.
*/
/* commented out for your protection
// We are not going to use the digraph method to calculate the
// JRC comparison scores.
// unsigned char digraph_score(const char *string1, const char *string2)
{
    char tempDigraphStr(2 * 11); // storage for a digraph string
    int stringLen = strlen(string1);
}

```

[illegible]

```

char ruleBuf(1000 + 1);
char *buffer;
int lineNo = 0;
int i, j;
char errMsg(1000 + 1);

m_rules.clear();
m_simplifiedRules.clear();
m_simplifiedRules.clear();

// create a vector of Strings for the replacement strings as read from
// the simplifiedRulesFile.
// Also, create a parallel vector of Strings for the simplified
// rules as read from the simplifiedRulesFile.
while ((fgets(ruleBuf, 1000, simplifiedRulesFile)) != rc)
{
    lineNo++;
    buffer = strtok(ruleBuf, "\n");
    // check for empty lines
    if (buffer != NULL)
    {
        if (buffer[0] == '\n')
        {
            buffer = strtok(NULL, "\n");
            if (buffer != NULL)
            {
                regStrings.push_back(buffer);
                buffer = strtok(NULL, "\n");
                if (buffer != NULL)
                {
                    simplifiedRegStrings.push_back(buffer);
                }
            }
        }
        else
        {
            sprintf(errMsg, "Invalid format on line %d of simplified rules file\n", lineNo);
            if (logFile != NULL)
            {
                AfxMessageBox(errMsg);
            }
            else
            {
                fprintf(logFile, errMsg);
            }
            rc = FALSE;
        }
    }
}

// }

if (rc)
{
    sprintf(errMsg, "Invalid format on line %d of simplified rules file\n", lineNo);
    if (logFile != NULL)
    {
        AfxMessageBox(errMsg);
    }
    else
    {
        fprintf(logFile, errMsg);
    }
    rc = FALSE;
}

// }

if (rc)
{
    // make sure that what is in the m_Rules vector is in the
    // regStrings vector. We can have things in the
    // regStrings vector that are not in m_Rules vector, since the
    // regStrings vector includes replacement strings for all
    // cultures, and this rule set is only for one culture.
    // Build a vector that is parallel to the m_Rules vector
    // that contains the simplified regex for each rule. This will
    // be similar to the simplifiedRegStrings vector, but will
    // be adjusted to be in the same order (and the same size as)
    // the m_Rules vector.

    CString ruleRegString;
    BOOL foundRegString;
    namSimplifiedRules = regStrings.size();
}

```

```

}
}

// now iterate through the aliases set and print out
// and aliases
set<CString>::iterator aliasesSetIter;
for (aliasesSetIter = aliases.begin(); aliasesSetIter != aliases.end(); aliasesSetIter++)
{
    sprintf(outBuf, "%s-10.100 ", (LPCTSTR)aliasesSetIter);
    outfile << (LPCTSTR)outBuf;
}
outfile << endl;

}
}

outfile << endl;
}

outfile << endl;
}

outfile << "Number of Rules: " << m_rules.size() << endl;
outfile << "Number of Disjunct Output Strings: " << m_namDisjunctStrings << endl;
outfile << "Number of Codes: " << m_namDisjunctCodes << endl;

return rc;
}

BOOL RuleSet::dumpToComparisonArray(outFile)
{
    BOOL rc = TRUE;
    char outBuf(1000);
    CString regBuf1;
    CString regBuf2;

    outfile << "Code\\Rule\\DisjunctOutputString\\DisjunctCode" << endl;
    for (int i = 1; i <= m_namDisjunctCodes; i++)
    {
        regBuf1 = m_namDisjunctCodes[i];
        for (int j = 1; j <= m_namDisjunctCodes; j++)
        {
            regBuf2 = m_namDisjunctCodes[j];
            sprintf(outBuf, "%d\\%d\\%d\\%d-10.100\\%d\\%d\\%d\\%d",
                i, j, regBuf1, regBuf2, m_namDisjunctCodes[i],
                m_namDisjunctCodes[j]);
            outfile << (LPCTSTR)outBuf << endl;
        }
    }

    return rc;
}

// }

}

function to read the supplied file of simplified rules and
1.) Ensure that the replacement strings in the simplified
rules file match the replacement strings of the rules file.
2.) Create a vector of codes (unsigned char).
The inner vector contains the codes corresponding to a
particular rule. The outer vector contains the inner
vectors, and corresponds (should be in the same order as
the vector of rules to the m_Rules vector.

Note That the logfile parameter can be NULL, in which case the error
messages are displayed via AfxMessageBox.
Also, the encodeRulesFile parameter can be NULL if the final list
of code/expression pairs is not needed.
RuleSet::addSimplifiedRules(FILE *simplifiedRulesFile, FILE *logfile, FILE *encodeRulesFile)
{
    BOOL rc = TRUE;
    int namRules = m_Rules.size();
    int namSimplifiedRules;
    vector<CString> regStrings;
    vector<CString> simplifiedRegStrings;
    // from simplifiedRulesFile
    // from simplifiedRulesFile
}

```



```

for (i = 0; i < numRules; i++) {
    ruleDescr = m_Rules[i].output;
    // skip empty replacement strings, since we want to just
    // ignore atoms. However, we still need to add the empty string
    // to the simplified strings vector so that it remains the
    // same size as the rules vector. We just skip the integrity check.
    if (ruleDescr.getLength() == 0) {
        foundDescr = FALSE;
        for (j = 0; j < numSimplifiedRules; j++) {
            if (ruleDescr == regStrings[j]) {
                foundDescr = TRUE;
                m_simplifiedDescrStrings.push_back(simplifiedStrings[j]);
                break;
            }
        }
        if (foundDescr == FALSE) {
            sprintf(errMsg, "Rule %d, Replacement String %s was not found in the
                simplified rules file\n",
                i, (LACTN)ruleDescr);
        }
    }
    else {
        // just a dummy value so that m_simplifiedDescrStrings remains para
        m_simplifiedDescrStrings.push_back("");
    }
}

// If all is ok so far, go through each of the m_simplifiedDescrStrings,
// and parse apart the simplified regular expression for the replacement
// portion of the rule. For each single-character expression in the
// simplified Rep String, add it to a map (unless it is already there).
// using an unsigned char code (for values that are
// optional, we use codes above 128). Either way, add that code
// to a temporary vector (which holds the codes for this simplified
// rep). At the end of the simplified regular expression, add the
// temporary vector to the m_simpCodeVec, which corresponds to
// the m_Rules vector, and holds a simplified code vector for each rule.
if (rc) {
    (LACTNString, unsigned char*: iterator codeMapIterator;
    char *repString(100 + 1);
    char *expStrPtr;
    BOOL optionalFlag;
    unsigned char optionalCodeIdx = 129;
    unsigned char mandatoryCodeIdx = 1;
    vector<unsigned char*> tempCodeVec;

    for (j = 0; j < numRules; j++) {
        tempCodeVec.clear();
        strcpy(repString, (LACTN)m_simplifiedDescrStrings[j]);
        if (*repString == '(') {
            if (*repString+1 == -1) {
                if (*repString+2 == -1) {
                    if (*repString+3 == -1) {
                        if (*repString+4 == -1) {
                            if (*repString+5 == -1) {
                                if (*repString+6 == -1) {

```



```

        if (m_weighted_score < 0.0f)
            m_weighted_score = 0.0f;
    }
    catch (...) {
        ASSERT(0);
        m_weighted_score = 0.0f;
    }
}

/* ..... */
/* phonetic_score
Returns a value in [0.0, 1.0] representing the value which was calculated
by the calling program.
*/
static const float phonetic_score(const float score)
{
    return score;
} // phonetic_score

/* culture_score
Returns a value in [0.0, 1.0] representing a penalty for when the
culture of the query and database names don't match.
*/
// narrative paragraph number 4.4.6
static const float culture_score(const float culture1, const float culture2)
{
    float rc;

    rc = culture1 == culture2 ? 1.0f : 0.0f;

    return rc;
} // culture_score

/* lead_cons_score
Returns a value in [0.0, 1.0] representing a scale of the minimum to
maximum phonetic-features distance which separate any of the leading
components of the two names. 0.0 is very far away; 1.0 is real close.
*/
// narrative paragraph number 4.4.4
static const float lead_cons_score(const byte *cons1_in, const byte *cons2_in,
                                   const unsigned int *featureDistanceTable)
{
    float rc = 0.0f; // very far away
    float min_diff = 1.0f; // very far away since this is the complement

    for (byte *p = const_cast<byte*>(cons1_in); *p != '\0'; p++)
        for (byte *q = const_cast<byte*>(cons2_in); *q != '\0'; q++)
            if ((*p) != (*q) && featureDistanceTable[*p] < min_diff)
                min_diff = (*p) && featureDistanceTable[*q];

    if (0.0f <= min_diff && min_diff <= 1.0f)
        rc = 1.0f - min_diff;

    return rc;
} // lead_cons_score

/* ..... */
/* spell_score
Returns a value in [0.0, 1.0] representing a closeness in the spellings
of the two names, and is calculated based on the number of bigrams and
single characters which match between the two given strings.
A bias can be used so that matches on the right end of the strings count
more than matches on the left end.
*/

```

```

    }
    catch (...) {
        ASSERT(0);
        return; // not attempting to do anything else right now
    }
}

/* ..... */
// other helper functions. These must return values in the range [0.0, 1.0].
static const float phonetic_score(const float);
static const float culture_score(const float culture1, const float culture2);
static const float lead_cons_score(const byte *, const byte *,
                                   const unsigned int *featureDistanceTable);
static const float spell_score(const char *, const char *,
                               const bool, const char, const char,
                               const float score, const float min_score);
static const float syll_score(const char *, const int);
static const float vowel_score(const char *, const int);
static const float vowel_index(const char *, const int);
static const float vowel_index(const char *, const int);
static const float vowel_index(const char *, const int);

/* ..... */
/* score
Calculate a weighted average of all voters.
Most of the voters will require additional calculations done by the
above helper routines.
*/
void Ranker::score(const Ranker *query, const Ranker *params,
                  const int query_syllables,
                  const unsigned int *featureDistanceTable)
{
    try {
        m_phonetic_score = phonetic_score(m_phonetic_score);
        m_culture_score = culture_score(query->getCulture(), params->getCulture());
        m_lead_cons_score = lead_cons_score(query->getLeadCons(), params->getLeadCons(),
                                           featureDistanceTable);
        m_spell_score = spell_score(query->getSpell(), params->getSpell(),
                                    query->getVowelIndex(), params->getVowelIndex(),
                                    query->getSyllableIndex(), params->getSyllableIndex());
        m_syllable_score = syll_score(query->getSyllable(), params->getSyllable(),
                                       query->getVowelIndex(), params->getVowelIndex());
        m_vowel_score = vowel_score(query->getVowel(), params->getVowel(),
                                    query->getVowelIndex(), params->getVowelIndex());

        // narrative paragraph number 4.4.7
        // There appears to be a bug in the optimizer-the "volatiles" fix it for now.
        volatile float numerator =
            * params.getPhonetic() * m_phonetic_score
            * params.getCulture() * m_culture_score
            * params.getLeadCons() * m_lead_cons_score
            * params.getSpell() * m_spell_score
            * params.getSyllable() * m_syllable_score
            * params.getVowel() * m_vowel_score;

        volatile float denominator =
            * params.getPhonetic()
            * params.getCulture()
            * params.getLeadCons()
            * params.getSpell()
            * params.getSyllable()
            * params.getVowel();

        m_weighted_score = numerator / denominator;
    }
    catch (...) {
        ASSERT(0);
        return;
    }
}

```

```

less than those on the left.
The score will possibly be modified if this name has a leading vowel.
*/
// narrative paragraph number 4.4.1
// narrative paragraph number 4.4.2
#pragma optimize("g", off) // there appears to be a bug in this optimisation
static const float spell_score(const char *str1_in, const char *str2_in,
    const bool bias,
    const char lead_vowel_1, const char lead_vowel_2,
    float mono_score, float bi_score)
{
    #ifndef BLANK
        static const char BLANK = ' ';
    #endif

    char str1[RANKER_NAME_SIZE * 2 + 1];
    char str2[RANKER_NAME_SIZE * 2 + 1];
    str1[0] = str2[0] = BLANK;
    strcpy(str1 + 1, str1_in);
    strcpy(str2 + 1, str2_in);
    strcat(str1, " ");
    strcat(str2, " ");
    unsigned int str1_len = strlen(str1);
    unsigned int str2_len = strlen(str2);
    char temp_str1[3];
    temp_str2[2] = '\0';

    // These are the weights a name has when using a biased
    // (left-to-right) calculation.
    // 1.0, 1.0, 1.9, 2.7, 3.4, 4.0, 4.5, 4.9, 5.2, 5.4, 5.5, 5.6, 5.7,
    // 5.8, 5.9, 6.0, 6.1, 6.2, 6.3, 6.4, 6.5, 6.6, 6.7, 6.8, 6.9, 7.0, ...
    static float weights[RANKER_NAME_SIZE * 2];
    // These three values need to be changed and chosen carefully-make
    // certain you understand the theory first!!
    // Examples: (0.1, 10, 0.1), (0.3, 4, 0.1), (0.2, 5, 0.2), (0.2, 4, 0.4), (0.1, 5, 0.5)
    static const float DECK = 0.1f;
    static const int MAXLEN = 5;
    static const float MIN_DVAL = 0.5f;
    static bool first_time = true;
    if (first_time) {
        weights[0] = weights[1] = 1.0f;
        for (int i = 2; i < sizeof(weights) / sizeof(weights[0]); i += MAXLEN; i++)
            weights[i] = weights[i - 1] * DECK * (1 - 1);
        for (i = 1 < sizeof(weights) / sizeof(weights[0]); i++)
            weights[i] = weights[i - 1] * MIN_DVAL;
        first_time = false;
    }

    bool bi_already_matched[RANKER_NAME_SIZE * 2]; // max bigrams = name size + 1
    bool no_already_matched[RANKER_NAME_SIZE * 2]; // max monograms = name size

    // Forget all matches.
    memset(bi_already_matched, false, sizeof bi_already_matched);
    memset(no_already_matched, false, sizeof no_already_matched);

    // Now count the number of elements involved in matching.
    float bias_q = 1.0f;
    float no_bias_db = 1.0f;
    bool no_compensate;
    float bi_match_count = 0.0f;
    float no_match_count = 0.0f;
    char *q;
    no_compensate = bias;
    for (unsigned int qk = 0; qk < str1_len - 1; ++qk) {
        // see if this digraph occurs in the database name
        temp_str1[0] = str1[qk];
        temp_str1[1] = str1[qk + 1];
    }
}

```

RANKER.CPP 3-24-98 11:24a

```

if (bias) { // decrement match-bias
    bias_q = 1.0 - DECK * qk;
    if (bias_q < MIN_DVAL) { bias_q = MIN_DVAL; no_compensate = false; }
}
// for bigrams
q = str2;
do {
    if (q != NULL) {
        unsigned int dx = static_cast<unsigned int>(q - str2);
        if (bi_already_matched[dx]) {
            bi_already_matched[dx] = true;
            bias_db = 1.0 - DECK * dx;
            if (bias_db < MIN_DVAL) bias_db = MIN_DVAL;
        }
        bi_match_count += bias_q * bias_db;
        break;
    }
    else
        q++;
}
while (q != NULL);
// for monograms
if (qk == 1) // ignore the leading and trailing blanks
    for (q = str2 + 1; q + 1 != '\0'; q++)
        if (qk == str1[qk]) {
            unsigned int dx = static_cast<unsigned int>(q - (str2 + 1));
            if (no_already_matched[dx]) {
                no_already_matched[dx] = true;
                if (bias) { // decrement match-bias
                    bias_db = 1.0 - DECK * dx;
                    if (bias_db < MIN_DVAL) bias_db = MIN_DVAL;
                }
                no_match_count += bias_q * no_bias_db * no_compensate ? DECK : 0.0f * bias;
            }
            break;
        }
    }
}
// Consider oneal vs. neal; change to oneal vs. xneal, where x matches anything.
// but it only gets 0.5 point instead of 1. Then onen => 2(0.5) plus
// bombx => 2(0.5), thus we add two to the numerator. For the denominator,
// it's like adding one more possible digraph (not two). This is only for
// bigrams.
int m = 0, n = 0;
if (lead_vowel_1 != R_NAME_1s lead_vowel_2 == R_NAME_1s
    || lead_vowel_1 == R_NAME_1s lead_vowel_2 != R_NAME_1s) {
    m = 2; // = 2(0.5) * 2(0.5)
    n = 1; // one more digraph
}
// The return value is the number of elements involved in matching
// compared to the local number of elements.
bi_score = bias
    ? (bi_match_count * m) / (weights[str1_len - 1] + weights[str2_len - 1] * n)
    : (bi_match_count * m) / (str1_len - 1 + str2_len - 1 * n);
mono_score = bias
    ? no_match_count / (weights[str1_len - 2] + weights[str2_len - 2])
    : no_match_count / (str1_len - 2 + str2_len - 2);
return (2.0 * bi_score + mono_score) / 3.0; // a 2:1 ratio
} /* spell_score */
#pragma optimize("g", on)

```

Page 3 of 5

[illegible]

```

m_lead_cons(0) = 1.0f;
m_lead_vowel = R_NONE;
m_is_exact = false;

m_phonetic_score = 0.0f;
m_culture_score = 0.0f;
m_lead_cons_score = 0.0f;
m_spelling1_score = 0.0f;
m_spelling2_score = 0.0f;
m_syllable_score = 0.0f;
m_vowel_score = 0.0f;
m_weighted_score = 0.0f;

}

Rhyme::Rhyme(const string &n, const e_tds_culture classOut,
              const e_tds_culture pipeOut,
              const float ps,
              const byte c1, const char v, const bool e)
{
    Rhyme(); // sets scores to zeros

    m_name_str = n;
    m_classOut = classOut;
    m_pipeOut = pipeOut;
    m_phonetic_score = ps;
    safety(m_lead_cons, c, R_MAX_LEAD_CONS);
    m_lead_vowel = v;
    m_is_exact = e;
}

void Rhyme::assign(const Rhyme &c)
{
    m_name_str = c.m_name_str;
    m_classOut = c.m_classOut;
    m_pipeOut = c.m_pipeOut;
    safety(m_lead_cons, c.m_lead_cons, R_MAX_LEAD_CONS);
    m_lead_vowel = c.m_lead_vowel;
    m_is_exact = c.m_is_exact;
    m_phonetic_score = c.m_phonetic_score;
    m_culture_score = c.m_culture_score;
    m_lead_cons_score = c.m_lead_cons_score;
    m_spelling1_score = c.m_spelling1_score;
    m_spelling2_score = c.m_spelling2_score;
    m_syllable_score = c.m_syllable_score;
    m_vowel_score = c.m_vowel_score;
    m_weighted_score = c.m_weighted_score;
}

// narrative paragraph number 4.4.8
bool Rhyme::operator==(const Rhyme &c) const
{
    if (m_is_exact && !c.m_is_exact)
        return false;
    if (m_is_exact && c.m_is_exact)
        return true;
    return m_weighted_score < c.m_weighted_score;
}

bool Rhyme::operator!=(const Rhyme &c) const
{
    return m_weighted_score == c.m_weighted_score;
}

```

```

void Rhyme::makeOppos() {
    char temp_str[BUFSIZ*2+1];
    strcpy(temp_str, m_name_str.c_str());
    strcpy(temp_str, m_name_str.c_str());
    m_name_str = temp_str;
}

bool less_Rhyme::operator()(const Rhyme *m1, const Rhyme *m2) const
{
    if (m1->getIsExact() && !m2->getIsExact())
        return false;
    if (!m1->getIsExact() && m2->getIsExact())
        return true;
    return m1->getWeightedScore() < m2->getWeightedScore();
}

/* ..... */
/* ..... */

```

```

// CSimpCodeDistanceTable implementation file
// The Simple Code Feature Distance Table class
// CSimpCodeDistanceTable is dependent on the
// CFeatureDistanceTable class
// the simple codes are split into two groups
// non-optimal codes have values [1,128]
// optimal codes have values [129,255]
#include "stdafx.h"
#include "simpcodeDistanceTable.h"
#include "defines.h"
#include "resource.h"

CSimpCodeDistanceTable::CSimpCodeDistanceTable(CFeatureDistanceTable *fctable,
simp_codes_map *symbols)
{
    init_table();
    calc_values(fctable, symbols);
    //amp();
}

CSimpCodeDistanceTable::CSimpCodeDistanceTable()
{
}

void CSimpCodeDistanceTable::init_table()
{
    int i, j;
    // initialize the insertion col (col 0) to 10,000
    for(i = 0; i < 256; i++)
        m_simp_code_dist_table[i][0] = 10000;
    for(j = 0; j < 256; j++)
        m_simp_code_dist_table[0][j] = 10000;
    // initialize the optional deletion part of the row to 0
    for(i = 129; i < 256; i++)
        m_simp_code_dist_table[i][0] = 0;
    // initialize everything else to -1,000,000. The cells
    // corresponding to used codes will be reset to a value
    // other than -1,000,000 later. Positive integer between 0 and
    // 10,000. The -1,000,000 will be used as a validity check if
    // necessary
    for(i = 1; i < 256; i++)
        for(j = 1; j < 256; j++)
            m_simp_code_dist_table[i][j] = -1000000;
}

void CSimpCodeDistanceTable::calc_values(CFeatureDistanceTable *fctable, simp_codes_map *symbols)
{
    CString key_string_scan_string;
    int act_i, act_j; // these index into the symbol code table
    simp_codes_map::iterator main_symbolit;
    simp_codes_map::iterator scanner_symbolit;
    int lowest_value;
    for(main_symbolit = symbols->begin(); main_symbolit != symbols->end(); main_symbolit++)

```



```

scan_length = j2;
for(i1 = 0; i1 < key_length; i1++)
{
    for(j1 = 0; j1 < scan_length; j1++)
    {
        fdist = 10000 * (ftable->get_feature_distance(key_buff(i1), scan_buff(j1)));
        if(fdist < lowest)
            lowest = fdist;
    }
}
return lowest;
}

void CSimpCodeHashTable::dump()
{
    int i,j;
    FILE *scamp;
    char msg[255];

    if((scamp = fopen("simpdump.txt","w")) == NULL)
    {
        sprintf(msg, "Could not open simpdump.txt for writing");
        MessageBox(msg);
        return;
    }

    for(i = 0; i < 256; i++)
    {
        for(j = 0; j < 256; j++)
        {
            fprintf(scamp, "%d %d simp_code_dist_table (%i %i)",
                i, j, scamp, "n");
            printf(scamp, "\n");
        }
    }
    fclose(scamp);
}

```

```

int lowest = 1000000;
int i1,i2,length,fdist;
unsigned char buff[10];

i1 = i2 = 0;

length = strlen(string);

for(i1 = 0; i1 < 10; i1++)
    buff[i1] = NULL;

// strip out { and ?
for(i1 = 0; i1 < length; i1++)
    if(string[i1] != '{' && string[i1] != '?')
    {
        buff[i2] = string[i1];
        i2++;
    }

length = i2;

for(i1 = 0; i1 < length; i1++)
{
    fdist = 10000 * (ftable->get_feature_distance(unsigned char(NULL), buff(i1)));
    if(fdist < lowest)
        lowest = fdist;
}

return lowest;
}

int CSimpCodeHashTable::calc_lowest_score(const char *key_string,
const char *scan_string,
CFeatureDistanceTable *f_table)
{
    int i; // generic counter
    int lowest = 1000000;
    int fdist;
    unsigned char key_buff[10];
    unsigned char scan_buff[10];
    int i1,i2,i3;
    int key_length = strlen(key_string);
    int scan_length = strlen(scan_string);

    i1 = j1 = i2 = j2 = 0;

    for(i = 0; i < 10; i++)
    {
        key_buff[i1] = NULL;
        scan_buff[i1] = NULL;
    }

    // strip out pipes (ors) and question marks from strings
    for(i1 = 0; i1 < key_length; i1++)
        if(key_string[i1] != '|' && key_string[i1] != '?')
        {
            key_buff[i2] = key_string[i1];
            i2++;
        }

    key_length = i2;

    for(j1 = 0; j1 < scan_length; j1++)
        if(scan_string[j1] != '|' && scan_string[j1] != '?')
        {
            scan_buff[j2] = scan_string[j1];
            j2++;
        }
}

```

```

// Copyright (C) 1998, Language Analysis Systems Inc.
//
// supplied.cpp: implementation of the CSimpleDistance class.
//
#include "stdafx.h"
#include "SimpleDistance.h"

// Construction/Destruction

CSimpleDistance::CSimpleDistance()
{
    //Initialize distance array to all 0's
    int i = 0; j = 0;
    for (i = 0; i < (STRLENMAX+1); i++)
    {
        for (j = 0; j < (STRLENMAX+1); j++)
        {
            distancearray[i][j] = 0;
        }
    }

    //Initialize first row and column to ascending integers
    for (i = 0; i < (STRLENMAX+1); i++)
    {
        distancearray[i][0] = i;
        distancearray[0][i] = i;
    }

    threshold = 0;
}

CSimpleDistance::~CSimpleDistance()
{
}

// Member functions

void CSimpleDistance::setThreshold(float inThreshold)
{
    threshold = inThreshold;
}

float CSimpleDistance::getThreshold()
{
    return(threshold);
}

// float CSimpleDistance::getDistanceScore(const char *set,
// **)
{
    int i;
    int j;
    int size;
    int size2;
    char str1[STRLENMAX+1];
    char str2[STRLENMAX+1];
    int diff;
    float lowest;
    float modDiff;
    float cutoff;
}

```

```

    {
        //coming from the left
        lowest = 1 + distanceArray[i-1][j-1];
        //coming from above
        diff = 1 + distanceArray[i-1][j];
        if (diff < lowest)
            lowest = diff;
        //coming from above left
        if (str1[i] == str2[j])
            diff = distanceArray[i-1][j-1];
        else
            diff = 1 + distanceArray[i-1][j-1];
        if (diff < lowest)
            lowest = diff;
        distanceArray[i][j] = lowest;
        if (lowest < curDiff)
            curDiff = lowest;
    }
    if (curDiff > modDiff)
        return(0.0);
    i++;
}
distance = distanceArray[i-1][j-1];
return 1.0f - (distance / (float)maxSize);
}

//
}

float ComputeDistance (getDistanceScore(const char *str1,
                                     const char *str2)
{
    int i;
    int j;
    int size1;
    int size2;
    const char *str1;
    const char *str2;
    const char *str1Start;
    const char *str2Start;
    int diff;
    int lowest;
    int modSize;
    int distance;
    float modDiff;

    //get size of str1
    size1 = strlen(str1);
    if (size1 > STR1MAX)
        return(-1);

    //get size of str2
    size2 = strlen(str2);
    if (size2 > STR2MAX)
        return(-2);

    // added when we removed the early out. since we
    // still need to assign str1Start, str2Start and
    // the modSize
    modSize = (size1 > size2) ? size1 : size2;

    // simple early out that just looks at length
    modDiff = (1 - (float)modSize) * 100;
    if (modSize1 - size2) > modDiff)

```

```

    return 0.0;

    str1Start = str1;
    str2Start = str2;

    //compute rest of array starting with second row
    i = 1;
    while (*str1 != EOS)
    {
        j = 1;
        strJ = str2Start;
        while (*strJ != EOS)
        {
            //coming from the left
            lowest = 1 + distanceArray[i-1][j-1];
            //coming from above
            diff = 1 + distanceArray[i-1][j];
            if (diff < lowest)
                lowest = diff;
            //coming from above left
            if (*strI == *strJ)
                diff = distanceArray[i-1][j-1];
            else
                diff = 1 + distanceArray[i-1][j-1];
            if (diff < lowest)
                lowest = diff;
            distanceArray[i][j] = lowest;
            strJ++;
        }
        strI++;
        i++;
        distance = distanceArray[i-1][j-1];
        return 1.0f - ((distance/((float)maxSize));
        return 1.0f - (distance / (float)maxSize);
    }
}

```

```
// Copyright (C) 1998, Language Analysis Systems Inc.  
//  
// stdafx.cpp : source file that includes just the standard includes  
// . CrustIndex.pch will be the pre-compiled header  
// . stdafx.obj will contain the pre-compiled type information  
//  
#include "stdafx.h"
```

File: DfUtil.cpp

Description:

Implementation of various utility functions used in the Soot

History:

5/15/97 BJB Created

#include <string.h>

#include "tbs_util.h"

```
// function to remove leading and trailing spaces from a string
// in place.
// Strip the string at either end or both ends.
// Strip the space until the character that should
// be stripped. We start by seeing if they want the
// trailing chars stripped, which is easy. We simply
// work backwards from the end of the string, looking for
// the first non-strippable character, and terminate the
// string just past that character. Then if they wanted
// leading chars stripped, we work forwards to the first
// non-strippable char, and then move that and each following
// char to the beginning of the string.
```

void tbs_strip(char *aString)

{ char *end_point;

char *ch;

int len;

if ((len = strlen(aString)) != 0) { // if there is a string

// start at end

end_point = aString + len - 1;

// and work back till we get a non-space or get to

// the beginning of our string, chopping off what's left.

// Also make sure we don't zoom right past the beginning of the

// string.

for (; strchr(TBS_DEFAULT_WHITESPACE, *end_point) != NULL && end_point != aString; end_point--

..)

// if string was all whitespace

if ((end_point == aString) && strchr(TBS_DEFAULT_WHITESPACE, *aString) != NULL)

*aString = EOS; // erase it all, and we're done, could return here

else

*(end_point + 1) = EOS; // just chop off excess blanks

// make sure there is still a string, since it might

// have been stripped entirely above.

if (*aString) {

// now find first non space. we know string has at least one

// nonwhite space, so we don't have to check for NULL.

for (ch = aString; strchr(TBS_DEFAULT_WHITESPACE, *ch) != NULL; ch++)

```
if (ch != aString) { // if there were leading spaces, move the block back
char *target = aString;
while (*ch != EOS) {
*target = *ch;
target++;
ch++;
} // and get the null char also
*target = '\0';
} // end if (are there leading spaces?)
} // end if (and text left?)
} // end (is there a string at all ?)
}

// function to do a string on unsigned chars
tbs_unsigned_strip(const unsigned char *s1, const unsigned char *s2)
{
while ((*s1 != EOS) || (*s2 != EOS)) {
if (*s1 != *s2) {
break;
}
else {
s1++;
s2++;
}
}
if (*s1 == *s2)
return 0;
else if (*s1 < *s2)
return -1;
else return 1;
}
}
```

class file for the TDSearcher class.

```
#include "cdutil.h"
#include "TDSearcher.h"
#include "Ranker.h"
#include "Distance.h"
#include "HandSecond.h"
```

typedef Ranker::ScoreBlanks::reverse_iterator cd_results_iterator;

```
// global error stream variable
ofstream err;
Distance soundDistance;
```

TDSearcher::TDSearcher(dstream *alogstream)

```
{
    status = false;
    simplifiedRulesFile = NULL;
    for (int i = 0; i < TDS_NUM_CULTURES; i++) {
        ruleset[i] = NULL;
        rulesetV[i] = NULL;
        namefiles[i] = NULL;
        groupdata[i] = NULL;
    }
    featureDistanceTable = NULL;
    simyCodeDistanceTable = NULL;
    editDistanceTable = NULL;
    simplifiedDistanceTable = NULL;
    featureEditDistanceTable = NULL;
    nameClassifier = NULL;
    falseVar = false;
    stopVar = falseVar;
    logstream = alogstream;
    logOutputInfo = false;
    queryStats.clear();
}

// for default, set to address of a variable set to false.
```

```
querySingleVowelVariants[0] = NULL;
querySingleVowelVariants[1] = NULL;
queryVowelVariants[0] = NULL;
queryVowelVariants[1] = NULL;

// Initialize the ranker weights to the defaults
rankerName = defaultRankerName;
multisetRankerQuery = TDS_DEFAULT_MAX_WEIGHTS_PER_QUERY;
nameEditDistanceThreshold = TDS_DEFAULT_NAME_EDITDIST_THRESHOLD;
groupEditDistanceThreshold = TDS_DEFAULT_GROUP_EDITDIST_THRESHOLD;
cultureCode = TDS_CULT_CODE_AUTO;
specificCulture = TDS_CULT_ANGLO;
preliminaryName = TDS_PP_NAME_FORK;
postliminaryName = TDS_PP_NAME_THREE;

// defaults for the fast ranker
fastRankerThreshold = rankerName - specThreshold;
fastRankerName = TDS_DEFAULT_MAX_WEIGHTS_PER_QUERY;

// which name is disabled by default
```

whichName[0] = '\0';

// initialize the arrays that hold the file names for the
// different data objects.

```
// rule files
rulesetFiles[TDS_CULT_ANGLO] = TDS_ANGLO_RULES_FILE;
rulesetFiles[TDS_CULT_ARABIC] = TDS_ARABIC_RULES_FILE;
rulesetFiles[TDS_CULT_CHINESE] = TDS_CHINESE_RULES_FILE;
rulesetFiles[TDS_CULT_HISPANIC] = TDS_HISPANIC_RULES_FILE;
```

```
// 3 vowel rules
rulesetVFiles[TDS_CULT_ANGLO] = TDS_ANGLO_3V_RULES_FILE;
rulesetVFiles[TDS_CULT_ARABIC] = TDS_ARABIC_3V_RULES_FILE;
rulesetVFiles[TDS_CULT_CHINESE] = TDS_CHINESE_3V_RULES_FILE;
rulesetVFiles[TDS_CULT_HISPANIC] = TDS_HISPANIC_3V_RULES_FILE;
```

```
// name files
namefiles[TDS_CULT_ANGLO] = TDS_ANGLO_NAME_FILE;
namefiles[TDS_CULT_ARABIC] = TDS_ARABIC_NAME_FILE;
namefiles[TDS_CULT_CHINESE] = TDS_CHINESE_NAME_FILE;
namefiles[TDS_CULT_HISPANIC] = TDS_HISPANIC_NAME_FILE;
```

```
// group files
groupfiles[TDS_CULT_ANGLO] = TDS_ANGLO_GROUP_FILE;
groupfiles[TDS_CULT_ARABIC] = TDS_ARABIC_GROUP_FILE;
groupfiles[TDS_CULT_CHINESE] = TDS_CHINESE_GROUP_FILE;
groupfiles[TDS_CULT_HISPANIC] = TDS_HISPANIC_GROUP_FILE;
```

```
// offsets (into name file) for the groups.
groupOffsetFiles[TDS_CULT_ANGLO] = TDS_ANGLO_GROUP_NAME_OFFSETS_FILE;
groupOffsetFiles[TDS_CULT_ARABIC] = TDS_ARABIC_GROUP_NAME_OFFSETS_FILE;
groupOffsetFiles[TDS_CULT_CHINESE] = TDS_CHINESE_GROUP_NAME_OFFSETS_FILE;
groupOffsetFiles[TDS_CULT_HISPANIC] = TDS_HISPANIC_GROUP_NAME_OFFSETS_FILE;
```

```
// culture strings
cultureStrings[TDS_CULT_ANGLO] = TDS_CULTURE_STRING_ANGLO;
cultureStrings[TDS_CULT_ARABIC] = TDS_CULTURE_STRING_ARABIC;
cultureStrings[TDS_CULT_CHINESE] = TDS_CULTURE_STRING_CHINESE;
cultureStrings[TDS_CULT_HISPANIC] = TDS_CULTURE_STRING_HISPANIC;
```

```
TDSearcher::TDSearcher()
{
    for (int i = 0; i < TDS_NUM_CULTURES; i++) {
        if (ruleset[i]) {
            delete ruleset[i];
            ruleset[i] = NULL;
        }
        if (rulesetV[i]) {
            delete rulesetV[i];
            rulesetV[i] = NULL;
        }
        if (namefiles[i] != NULL) {
            fclose(namefiles[i]);
            namefiles[i] = NULL;
        }
        if (groupdata[i]) {
            delete groupdata[i];
            groupdata[i] = NULL;
        }
        if (simplifiedRulesFile != NULL) {
            fclose(simplifiedRulesFile);
            if (featureDistanceTable != NULL)
                delete featureDistanceTable;
        }
    }
}
```



```

    rc = false;
    break;
}
else {
    // now try to encode the rules by handing it the simplified rules
    // file. If there are any rules in the rules file that have
    // replacement strings not found in the simplified rules file,
    // the function will put up a message box.
    if (ruleset(i) -> addSimplifiedRules(simplifiedRulesFile,
        NULL, /* log file, use message box instead */
        NULL /* code mapping file */ == false) {
        sprintf(errMsg, "Error adding simplified Rules to File (%s). Check to
        rules?
    }
    else {
        // g file".
        // loadRules(i);
        if (logstream != NULL)
            *logstream << errMsg << endl;
        rc = false;
        break;
    }
}
}
else {
    sprintf(errMsg, "Error opening simplified Rules File (%s)".
    TDS_SIMPLIFIED_RULES_FILE);
    if (logstream != NULL)
        *logstream << errMsg << endl;
    rc = false;
}
return rc;
}

bool TDS::Searcher::loadRules()
{
    bool rc = true;

    for (int i = 0; i < TDS_NUM_CULTURES; i++) {
        break;
        ruleset(i) = new Ruleset(rulesVFileNames(i));
        if (ruleset(i) -> Reader() == false) {
            sprintf(errMsg, "Error reading %d Vowel Rules File (%s)", rulesVFileNames(i));
            if (logstream != NULL)
                *logstream << errMsg << endl;
            rc = false;
            break;
        }
    }
    return rc;
}

bool TDS::Searcher::loadMetadata()
{
    bool rc = true;
}

```

```

    HAS_CHD_ON_DICOMPHS_FILE_NAME,
    HAS_CHD_ON_TRICOMPHS_FILE_NAME,
    HAS_CHD_SN_FILE_NAME,
    HAS_CHD_SN_DICOMPHS_FILE_NAME,
    HAS_CHD_SN_TRICOMPHS_FILE_NAME,

    HAS_HISP_ON_FILE_NAME,
    HAS_HISP_ON_DICOMPHS_FILE_NAME,
    HAS_HISP_ON_TRICOMPHS_FILE_NAME,
    HAS_HISP_SN_FILE_NAME,
    HAS_HISP_SN_DICOMPHS_FILE_NAME,
    HAS_HISP_SN_TRICOMPHS_FILE_NAME);

    if (nameClassifier->get_status() != 0) {
        rc = false;
        sprintf(errMsg, "Error initializing HAS Name Classification facility.");
        if (logstream != NULL)
            *logstream << errMsg << endl;
        break;
        // end of switch
    }
    if (rc && (*recipVar == false)) {
        status = true;
        if (logstream != NULL)
            *logstream << "TDSSearch object initialized successfully" << endl;
        return rc;
    }
}

bool TDS::Searcher::loadRules()
{
    bool rc = true;

    simplifiedRulesFile = fopen(TDS_SIMPLIFIED_RULES_FILE, "r");
    if (simplifiedRulesFile != NULL) {
        for (int i = 0; i < TDS_NUM_CULTURES; i++) {
            break;
            // make sure to rewind the file, since the addSimplifiedRules
            // method reads to the end
            rewind(simplifiedRulesFile);
            ruleset(i) = new Ruleset(rulesFileNames(i));
            if (ruleset(i) -> Reader() == false) {
                sprintf(errMsg, "Error reading Rules File (%s)", rulesFileNames(i));
                if (logstream != NULL)
                    *logstream << errMsg << endl;
            }
        }
    }
}

```



```

// BP adjustment via the simple vowel rules), and TDS BP_RULE_TABLES
// BP adjustment via the 3 vowel rules).
// The pre-ranker adjustments are pretty straightforward, since we can
// just check the value of the preRankerBiospec variable and call a
// brute force calculation right before a name is submitted to the Ranker.
// However, the post-ranker operation is more complicated, because it
// involves the use of two Rankers - a Fat Ranker that gets the initial
// pool, and a final ranker that received the names from the Fat Ranker
// (with adjusted edit distance scores). The Fat Ranker is initialized
// with a different threshold (usually lower) and max names value (usually
// higher) since we do not want to squeeze out any names that come
// out relatively high after their edit distance is adjusted.
// We chose the design of a Fat Ranker and Final Ranker so that the Ranker
// does not need to be concerned (have knowledge of) the BP Edit Distance
// calculation and all of the messy stuff that goes along with it (Rules, etc).

// clear out the Ranker object
ranker.Init(BcRankName(), BcRankName(), *rankerParams, rankerEditDistanceQuery,
           featureDistanceTable);

// get rid of the NR's data for the query name (2 culture)
queryInfo[0].RemoveAll();
queryInfo[1].RemoveAll();

// empty out all the data we collected on the query
emptyRankmap();
eachCulturalNameOffsets.clear();
queryGroups[0].clear();
queryGroups[1].clear();
if (querySingleBowlVariants[0] != NULL) {
    delete querySingleBowlVariants[0];
    querySingleBowlVariants[0] = NULL;
}
if (querySingleBowlVariants[1] != NULL) {
    delete querySingleBowlVariants[1];
    querySingleBowlVariants[1] = NULL;
}
if (queryVowelVariants[0] != NULL) {
    delete queryVowelVariants[0];
    queryVowelVariants[0] = NULL;
}
if (queryVowelVariants[1] != NULL) {
    delete queryVowelVariants[1];
    queryVowelVariants[1] = NULL;
}

// clear out the query stats structure
queryStats.clear();

// narrative parameter number 4.1.1
if (frc - validateQueryName(qname) == true) {
    // here we need to examine the culture setting the
    // user has specified on the GUI. If they have said
    // they want a particular culture, we should search that
    // culture and ANGLIO.
    // If they did not specifically request a culture, we should
    // search ANGLIO and the culture the classifier comes up with for
    // the name. Since the classifier might decide the name is
    // not any of the other culture, we might end up searching
    // only ANGLIO.
    nomenclatureSearch = 1;
    queryCultures[0] = TDS_CULT_ANGLIO; // initialise
    // set to ANGLIO always, since we always s
}

```


Page 6 of 18

TOSSEA-1.CPP 3-24-98 11:24a

```

>> i;
    @_db_culture
    set-unsigned int>
    set-unsigned int>; iterator
        *eachMatchCandidateOffsets;
        culture;
        *eachMatchCandidateOffsets;

if ($logstream != NULL)
    $logstream << endl << "Starting Exact Search" << endl;

for ($i = 0; $i < numCulturesSearch; i++) {
    culture = queryCultures[$i];

    narrative paragraph number 4.2.1
    // Ask the GroupData object for the offsets of names that produce
    // a group that matches one of the query's groups exactly.
    // Offsets into the NamesSet file.
    eachMatchCandidateOffsets =
        groupData[culture]-getNamesOffsetsOfExactGroupMatches(queryGroups[$i]);

    if (*scopVar) {
        delete exactMatchCandidateOffsets;
        break;
    }

    // store in the query info how many names were brought back because they
    // had a group that matched the query's group exactly.
    queryStats.namesWithExactMatchGroup($i) = exactMatchCandidateOffsets->size();

    // now go through each of the candidates and retrieve the data
    // at the specified offset. The function getMembersAndEvaluate will
    // add the name data to a map if it passes the exact match comparison
    for (eachMatchOffsetIter = exactMatchCandidateOffsets->begin();
        exactMatchOffsetIter != exactMatchCandidateOffsets->end();
        exactMatchOffsetIter++) {
        narrative paragraph number 4.2.2
            if (getMembersAndEvaluate(*exactMatchOffsetIter,
                culture,
                $i, // first or second culture to search? (0 or 1)
                THIS_FROM_EXACT_SEARCH) == false) {
                    rc = false;
                    break;
                }

                // got rid of the offsets, now that we have looked at them
                delete exactMatchCandidateOffsets;

                if (rc) {
                    if ($logstream != NULL) {
                        $logstream << "Culture " << cultureStrings[culture]
                            << "; Names that matched exact
                                << queryStats.namesThatMatchedExact";
                    }
                }
            }
        }
    }
}

```

```

// now we have done exact lookups for both cultures
// so we should go through the map of NameRecords and
// send the information to the Ranker.
// Each NameRecord in the map can represent 1 or two cultures.
// If a name represents two cultures, we need to send both
// sets of info (the separate NameRecord objects) to the Ranker
// so it can decide which one is better.
//
// Note that we use the final ranker here no matter what, because
// exact matches do not need to have their edit distance adjusted.
// Therefore we can go straight to the final ranker with them.
//
// narrative paragraph number 4.2.4
name_record_map::iterator
mapIter = nameRecord.begin();
for (mapIter = nameRecord.begin(); mapIter != nameRecord.end(); mapIter++) {
    //
}

if (!logStream) {
    if (rc == false)
        *logStream << "Processing stopped due to error" << endl;
    else
        *logStream << "Exact Match Search Completed" << endl;
}

return rc;
}

//
// function to do the second phase of the query. This involves
// searching through each record in the group file
// and performing an edit distance operation between that group and
// all the groups that the query generated. If any pass the
// edit distance threshold specified for the group comparison, we
// look at the names associated with that group. For each of those
// names, we perform an edit distance comparison one the SSC strings
// associated with the query name and the name in question.
//
// After we have done this for the cultures we are suppose to, we
// go through the NameRecord map, and tell each NameRecord to
// submit itself to the Ranker. However, since the NameRecord map
// also contains exact matches, we should make sure we skip those.
//
// Note that the code for this function is very similar to the
// doExactSearch, except that:
//
// We can skip alot of query name processing, since that was already done.
// We look at all the groups, rather than using a binary search.
// We we look at names, we do an edit distance rather than an exact phonetic
// match comparison.
//
// TOSSEARank::SearchForSimilarMatches()
bool
{
    bool
    rc = true;
    int
    i;
    e_cdu_culture
    secAssigned int;
    secAssigned int;
    int
    nameGroupThatPassedGroupDist;

    //
}

```

```

//
// Note that we use a different ranker depending on the value of the
// postNameRecord variable. If we are supposed to be adjusting the
// edit distance of the names after the ranker has worked on them,
// we should submit names to the Final Ranker. Otherwise, we should
// just add them to the final ranker.
// If (postNameRecord != TOS_PP_NAME_RECORD)
//     rankerToUse = finalRanker;
// else
//     rankerToUse = ranker;
//
// if (!logStream) {
//     *logStream << "Starting Similar Search" << endl;
//
//     ??? here is where we would calculate if the watch name should be
//     considered a similar match candidate based on its groups (Does any
//     of the groups the watch name produces match any of
//     the groups the query name produces with a score >= the threshold? )
//
//     for (i = 0; i < nameCulturesSearch; i++) {
//         culture = queryCultures[i];
//
//         // Ask the GroupData object for the offsets of names that produce
//         // a group that matches one of the query's groups exactly.
//         // offsets into the NameRec file.
//         narrative paragraph number 4.3.1
//         similarMatchCandidateOffsets =
//             groupData[culture].getNamesForGroups(queryGroup(i));
//
//         //
//         // up to that PassedGroupDist.
//         //
//         //
//         // score the query info for this portion of the search
//         queryStats.nameGroupThatPassedDist(i) = groupData[culture].getNamesForGroups(i);
//         queryStats.nameGroupThatPassedDist(i) = nameGroupThatPassedGroupDist;
//         queryStats.nameCandidateNamesFromSimilarSearch(i) = similarMatchCandidateOffsets - size
//
//         if (!stopVar) {
//             delete similarMatchCandidateOffsets;
//             break;
//         }
//
//         // now go through each of the candidates and retrieve the data
//         // at the specified offset. The function getNamesForGroups will
//         // add the name data to a map if it passes the exact match comparison
//         // for (similarMatchOffset = similarMatchCandidateOffsets - begin(i);
//             similarMatchOffset != similarMatchCandidateOffsets - end(i);
//             similarMatchOffset++) {
//             // narrative paragraph number 4.3.2
//             if (getNamesForGroups(*similarMatchOffset).
//
//         // culture.
//
//         // i, // first or second culture to search? (0 or 1)
//         // TOS_PP_NAME_RECORD == false) {
//             rc = false;
//         }
//     }
// }

```

Page 9 of 18

YDSSEA~1.CPP 3-24-98 11:24a

IDSSEAT1.CPP 3-24-98 11:24a

```

namesFilesOut;
bytestream = fread(char *nameBinaryBuf, 1, TDS_MAX_NAME_ENTRY_LEN);

** uell);
if (bytestream != TDS_MAX_NAME_ENTRY_LEN) {
    // its possible that we read the last entry (or near the last entry),
    // and that we reached the end of the file. Remember that we are
    // using a worst case estimate for the name record, since it's
    // length is variable. If we have reached the end of the file, we
    // should clear the eof flag so that future fread calls will work.
    // If we not have reached the end of file, then there is some real
    // problem.
    if (!feof(namesFiles(culture))) {
        clearer(namesFiles(culture));
    } else {
        sprintf(errmsg, "Error reading %d bytes at offset %d from names file %s".
            offSetInNames);
        rc = false;
        if (!logstream || !NULL) {
            *logstream << "Error Looking for a Name in the Names File" << endl;
            *logstream << errmsg << endl;
        }
    }
} else {
    sprintf(errmsg, "Error seeking to offset %d in names file %s".
        offSetInNameFile, namefileNumber);
    rc = false;
    if (!logstream || !NULL) {
        *logstream << "Error Looking for a Name in the Names File" << endl;
        *logstream << errmsg << endl;
    }
    return rc;
}

TDSSEARCHER::setGroupDistanceThreshold(float athresh)
{
    bool rc = true;
    if ((athresh > 1.0) || (athresh < 0.0)) {
        sprintf(errmsg, "Invalid New Edit Distance threshold %f", athresh);
        if (!logstream || !NULL)
            *logstream << errmsg << endl;
        rc = false;
    }
    nameEditDisctHresh = athresh;
    return rc;
}

TDSSEARCHER::setGroupDistanceThreshold(float athresh)
{
    bool rc = true;
    if ((athresh > 1.0) || (athresh < 0.0)) {
        sprintf(errmsg, "Invalid Group Edit Distance threshold %f", athresh);
        if (!logstream || !NULL)
            *logstream << errmsg << endl;
    }
}

```


**This Page is Inserted by IFW Indexing and Scanning
Operations and is not part of the Official Record**

BEST AVAILABLE IMAGES

Defective images within this document are accurate representations of the original documents submitted by the applicant.

Defects in the images include but are not limited to the items checked:

- ☐ **BLACK BORDERS**
- ☐ **IMAGE CUT OFF AT TOP, BOTTOM OR SIDES**
- ☐ **FADED TEXT OR DRAWING**
- ☐ **BLURRED OR ILLEGIBLE TEXT OR DRAWING**
- ☐ **SKEWED/SLANTED IMAGES**
- ☐ **COLOR OR BLACK AND WHITE PHOTOGRAPHS**
- ☐ **GRAY SCALE DOCUMENTS**
- ☐ **LINES OR MARKS ON ORIGINAL DOCUMENT**
- ☐ **REFERENCE(S) OR EXHIBIT(S) SUBMITTED ARE POOR QUALITY**
- ☐ **OTHER:** _____

IMAGES ARE BEST AVAILABLE COPY.

As rescanning these documents will not correct the image problems checked, please do not report these problems to the IFW Image Problem Mailbox.